App# 10/628959

JavaBeat

Java Server Faces(JSF) - Overview

JavaServer Faces is based on Specification Request JSR 127 released on 2004. Main purpose of it is to create a collection of APIs for the UI components managing their state together with event handling and validation. Custom tags allow it to be integrated into JSP pages.

As long as JavaServer Faces technology was developed under the Java Community Process it is a vendor independent technology representing a standard to be supported by whole software world. Main reason for it was to create one technology able to fit web-designers and component developers requirements into an easy-to-use way.

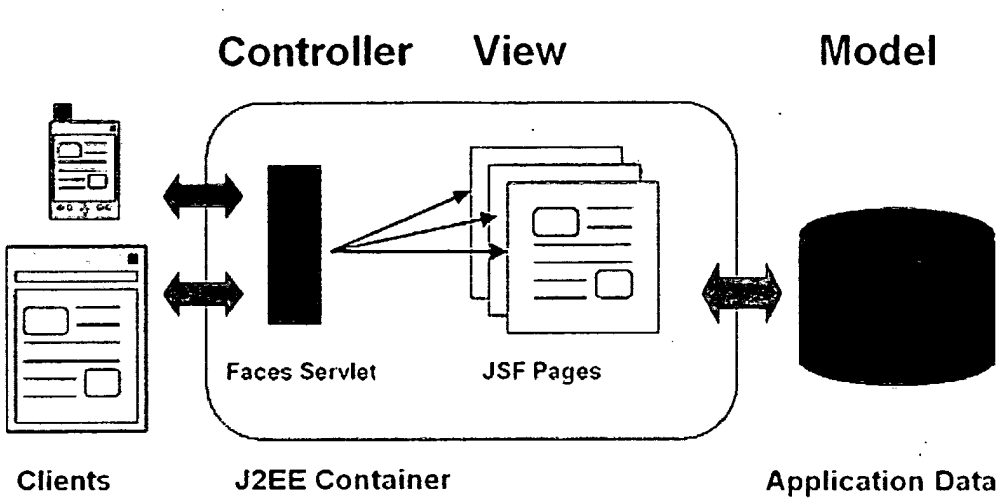
What is JSF?

JavaServer Faces (JSF) is a new standard Java framework for building Web applications. It simplifies development by providing a component-centric approach to developing Java Web user interfaces. JavaServer Faces also appeals to a diverse audience of Java/Web developers. "Corporate developers" and Web designers will find that JSF development can be as simple as dragging and dropping user interface (UI) components onto a page, while "systems developers" will find that the rich and robust JSF API offers them unsurpassed power and programming flexibility. JSF also ensures that applications are well designed with greater maintainability by integrating the well established Model-View-Controller (MVC) design pattern into it's architecture. Finally, since JSF is a Java standard developed through Java Community Process (JCP), development tools vendors are fully empowered to provide easy to use, visual, and productive develop environments for JavaServer Faces.

JSF Architecture

Official Sites

- JSF Home
- My Faces Home Page (Apache)



Resources

UI frameworks and JavaServer Faces
JSF Navigation by Examples
The Many Faces of JavaServer Faces (JSF)
Creating JSF Custom Components
Handling Events in JavaServer Faces, Part 1

Introduction to Java Server Faces

by Shunmuga Raja
19/05/2007

1) Introduction

Java Server Faces (JSF) simplifies the development of User Interfaces in a typical Web application. It provides a Component-based Pluggable Architecture for developing and representing User Interface Components. Also bundled with JSF is a whole set of various new features like Event Handling Mechanism, Page Navigation, Input Validation and Conversion. This article provides an overview about ***JSF Framework*** and the various core elements that form the basis for a ***JSF Framework***. It then provides a detailed overview about the various phases in the ***JSF Request Processing Life-cycle***. Explained further is the structure of the ***JSF Configuration File***. And finally the article is concluded by providing a sample application that demonstrates the various new features.

2) Java Server Faces - An UI Framework

In Fact, ***JSF*** is nothing but an abstraction over the existing Web Framework. ***JSF*** is implemented as a ***Servlet*** which is called the ***Faces Servlet***. Before the advent of ***JSF***, ***Servlets*** and ***JSP*** are the predominant ones that form the core components in the development of a Web Application. Let us see the traditional interaction that takes place in a Web Application is developed only with the ***Servlet*** and the ***JSP*** components that follows the ***MVC-2 Architecture***. A client who is normally a HTML Browser sends a request to the server. The ***Web Server*** receives the request, encapsulates the request and then populates this request object with the various parameter values from the client and will send it to the ***Servlet***. The Servlet which acts as a ***Controller***, analyses the request, then will interact with the ***Model (Java Beans)*** that executes the various application business logic and then chooses which ***View*** to be shown to the User.

Java Server Faces which provides a ***Component-Based Architecture*** for developing reusable ***User Interface Components*** hides most of the complex stuffs that are happening in the ***View portion*** of the ***MVC-2 Architecture***. The framework is not only limited to developing customized User Interface Components but also provides support for various Advanced Features like ***Event handling Mechanism***, ***Validating User Inputs*** that are sent by the clients, ***Easy Page Navigation Mechanism*** etc. The good thing about ***Java Server Faces*** is that the degree of coupling between the UI Components that represent the various ***behaviour/properties*** and its ***Rendering*** is very low. In fact it is almost nill. So, HTML browsers are not the only target client applications. JSF Applications works even well with WML Browsers.

3) Basic Elements of JSF

Let us examine the various core basic elements in a JSF application. The most significant components in a JSF application are explained as follows.

- User Interface Components
- Managed Beans
- Validators
- Convertors
- Events and Listeners
- Page Navigation
- Renderers

All the above-mentioned components can be found in separated packages in the API. For example, the ***User Interface Components*** are available in `javax.faces.component` package, ***Validation API*** is available in `javax.faces.validator` package and so on.

3.1) User Interface Components

If ***Java Swings*** represent the ***UI components*** for a ***Desktop Java Application***, then ***JSF UI Components***

**Latest Article
PlanetOSS.c**

SQLite with PHP 5.0
PHP, an acronym for
Preprocessor, is a ge
language that is use
pages.

SimpleXML in PHP 5.
PHP, an acronym for
Preprocessor, is a ge
language that is use
pages. Recently, a n
released and it is po
The top seven featur

Object Oriented Con
PHP, an acronym for
Preprocessor, is a sc
used for creating dyr
web pages. PHP is w
freely downloaded fr
http://www.php.net/

Securing a Web Page
Modules
The commands and t
article assumes that
following locations,

Creating customised
Preventing Hotlinkinc
This article discusses
Web Server namely
error messages and
Images.

are meant for the ***Web Applications***. The ***JSF User Interface Components*** are developed with ***Java Beans Specifications and Standards*** in mind. It means that ***JSF Components*** have ***properties***, ***methods*** and ***events*** as they are normally found for a traditional ***Java Bean***. One of the peculiar features of the ***JSF UI Components*** is that they can manage the ***state*** of the component. So many built-in UI components are bundled with the JSF API, the most commonly used ones are ***Label***, ***Text-Field***, ***Form***, ***Check-Box***, ***Drop-Down Box*** etc. JSF also provides a framework for creating ***Customized UI Components***.

It is very important to note that ***JSF UI Components*** only represent the ***attributes***, ***behaviors*** and ***events*** for a component and not the actual display. For example in a ***Text-Field Control***, the ***attributes*** may be the value of the text-field, the maximum number of characters that can be entered etc. Modifying the existing value and setting the maximum number of characters forms the ***behaviour*** for the Text-Field. Change in the value of the Text-Field may cause the Text-Field to cause some kind of ***Event*** to be emitted. The manner is which text-field is displayed in the client is separated from the ***UI Data Representation*** itself.

It means that there are separate components called ***Renderers*** which will take care of displaying the UI components in ***Different Client Surfaces***, one client may be the ***HTML Browser*** running in a PC and the other may be the ***WML Browser*** running within a mobile phone. Each and every JSF component is uniquely identified by a ***Component Identifier***.

3.2) Managed Beans

Managed Beans are standard Java classes that follow the ***Java Beans Specification***.Generally, ***Managed Beans*** are used to represent the user inputs. They may even act as ***Listeners*** and can handle the appropriate ***Actions***. Assume that there is an Encryption Application, which is presented with a Text-Field, wherein which user can enter a string that is to be encrypted. Below that is a Encrypt Button, which when clicked calls the server code that does the actual job of Encryption.

The following code snippet shows how to represent the Text-Field and the command button,

```
<html:inputText
    id = "strToBeEncryptedTextField"
    value = "#{EncryptionBean.strToBeEncrypted}">
</html:inputText>

<html:commandButton
    id = "encryptButton"
    actionListener = "#{EncryptionBean.doEncryption}">
</html:commandButton>
```

Don't worry about the declaration of the Text-Field and the Command Button here. The syntax for their declaration in covered in the later sections.

The first noticeable thing is that the Text-Field has two attributes namely ***id*** and ***value***. The id attribute uniquely identifies the Text-Field component from other Components in the View. The value for the attribute ***value*** is `#{EncryptionBean.strToBeEncrypted}`. This expression is a ***JSF EL Expression***. The EL expression can be interpreted as follows, Take the value of the string to be encrypted from the UI Component and map it to the property called `strToBeEncrypted` which is inside the class called `EncryptionBean`.

It means that the declaration of the ***Managed Java Bean*** class may look something like the following,

```
class EncryptionBean{

    private String strToBeEncrypted;

    public String getStrToBeEncrypted(){
        return strToBeencrypted;
    }

    public void setStrToBeEncrypted(String strToBeEncrypted){
        this.strToBeEncrypted = strToBeEncrypted;
    }
    // Other things goes here
}
```

From the declaration of the command button object, we can interpret that, 'encryptButton' is uniquely used to

identify the command button object, the attribute ***actionListener*** is used to provide the listener method that will get invoked as a result of someone clicking the button. So, `actionListener = "#{EncryptionBean.doEncryption}"` essentially says that there is a method called `doEncryption()` within the `EncryptionBean` class. Following code snippet may prove this,

```
class EncryptionBean{
        ...
        public void doEncryption(javax.faces.event.ActionEvent event){
        }
        ...
}
```

3.3) Validator

Validation is a must for almost any application. Data entered by the clients have to be validated before being sent to the Server for processing. ***JSF*** already have the ***Common Validation API*** being implemented for almost all controls in the `javax.faces.validator` package and also through various ***Custom Tags***. The ***Validator Framework*** that is available with ***JSF*** is pluggable, i.e it also allows the developers to provide their own ***Custom Validation Classes*** through the help of ***Configuration Files***.

Suppose say, there is a Text-Field Component in the form which inputs the age from the user to get some benefit from the organization. We can have a Simple ***Validation Rule*** telling that only Employees who are above 35 and below 45 are eligible for such a benefit. This can easily achieved through the use of ***Custom Validation Tags*** like the following one,

```
<html:inputText identifier = "employeeAgeTextField">
    <f:validateLongRange minimum = "25" maximum = "35">
    </f:validateLongRange>
</html:inputText>
```

JSF will immediately report a ***Default Error Message*** whenever the value of the age entered by the user crosses the boundary range of 25-35.

Two types of Validation are possible in JSF. They are,

- Direct Validation
- Delegated Validation

i). Direct Validation:

Assume that there is a ***Customized UI component*** called E-Mail Text Field which allows the user to enter only email-ids in the appropriate format. Whenever a user enters some email-id, the UI component has to validate whether the given email-id is in the right format, something like username@someDomain.com. It is wise to embed the ***Validation Logic*** within the component itself. Such Validation Code which is found within the UI component and can be used only by that Component is called ***Direct Validation***.

ii). Delegated Validation:

Delegation Validation comes into picture when the Validation Logic is about to be re-used across so many Components. ***Common Validation*** stuffs are ideal candidates for Delegated Validation. For example, consider in a form where we have a Text-Field representing the name of a customer and a Radio-Button for Marital Status with values 'Married' and 'Single'. If both are required fields, then before the submission of the form, both the input fields (text-field and the radio-button) have to be validated for empty (or null) values. So, in such a case we can have a Validator called `NullCheckValidator` and then bind both the input components to this Validator.

3.4) Convertors

Every piece of request that is passed from the client to the server is interpreted as a String value only. Manual conversion of the String object to the appropriate Data-type has to be done in the Application Code before carrying on with the application logic. Suppose in a form, there are fields like name, age and date. Corresponding to this form, we would have constructed a ***Managed Bean*** representing name, age and date as ***properties*** with String, integer and Date respectively. Certain amount of code has to be written for the conversion of the age and date values to their corresponding int and Date types.

But because of the availability of ***JSF Convertors*** functionality, this becomes easy. In the declaration of the ***UI Component*** itself within the form, we can mention which data-type the value for this control has to be converted. The ***Conversion API*** is available in `javax.faces.convert`.

For example, consider the following piece of code,

```
<html:inputText identifier = "numberTextField">
    <f:convertNumber pattern = "###,###">
    </f:convertNumber>
</html:inputText>
```

In the above, we have a ***Number Converter*** which converts the number given by the user to the specified format. For example, if the original value entered by the user is 123456, then the after the conversion process the value becomes '123,456'.

3.5) Events and Listeners

JSF UI Event Mechanism is very similar to the one found in ***Swing UI Components***. The Architecture remains the same in both the cases. In JSF, all the ***UI Components*** can emit any number of events. For example, a ***Button Component***, when clicked by the user can emit an `ActionEvent`. Those which take appropriate actions after a component has emitted some kind of Events are called ***Event Listeners***. In JSF, Java ***Managed Beans*** also acts as ***Listeners*** for the ***Events*** emitted by the components.

For example, consider the following code snippet,

```
<html:inputText
   identifier = "submitButton"
   value = "Click Me"
   actionListener = "#{SomeBean.submitButtonClicked}"
</html:inputText>
```

In the above code, we can see that how a ***UI Component*** can be associated with an ***Event-Handler*** with the help of actionListener attribute. The code essentially says that whenever an `ActionEvent` is emitted by the button (which will happen usually when the user clicks the button or presses the Enter key over the button) call the Event Listener's method `submitButtonClicked` inside the `SomeBean` class.

3.6) Navigation

An user of a Web Application doesn't restrict himself in viewing one single Web Page. He/She will ***navigate*** from one page to another page. Technically the navigation of a user from one page to another page results in the

generation of request and response for that page. Most of the boiler-plate work that is related to navigation stuffs in a web-application is handled by the ***Default JSF Navigation Handler*** itself. The Navigation Handler provides a simple yet a powerful framework for controlling the navigation.

For any single request page, there may be a number of response pages. Assuming that in a data-entry application, if the request page, say 'enterdata.jsp', is a view showing all the input controls to get data from the user, then the following responses may be available.

- The user has entered all the data and the result is a 'success' so that a page called 'success.jsp' page is displayed
- The user has entered a mix of correct and incorrect data in which case, the result or the response is a 'partialsuccess' and some jsp page called 'partialsuccess.jsp' is displayed.
- The user hasn't entered any correct values which means that result is a 'failure' which results in the display of a page called 'failure.jsp'.

If we look at a very high-level, almost all of the pages will have this kind of ***Navigation Rule*** which will have N number of ***Navigational Cases***. As such, we can define the Navigational Rules along with Navigational Cases for handling the navigational logic for the entire JSF application in the ***JSF Configuration File***.

For example, the following is a simple navigation rule for the above sample scenario,

```
<navigation-rule>
    <from-view-id>/dataentry/enterdata.jsp</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/dataentry/success.jsp</to-view-id>
        </navigation-case>

        <navigation-case>
            <from-outcome>partialsuccess</from-outcome>
            <to-view-id>/dataentry/partialsuccess.jsp</to-view-id>
        </navigation-case>

        <navigation-case>
            <from-outcome>failure</from-outcome>
            <to-view-id>/dataentry/failure.jsp</to-view-id>
        </navigation-case>
</navigation-rule>
```

The above code within the Configuration File states for the jsp file 'enterdata.jsp' (where '/dataentry/' is the context path), if the outcome is 'success', navigate to 'success.jsp, if the outcome is 'partialsuccess', then navigate to 'partialsuccess.jsp', else if the outcome is 'failure', then take the user to 'failure.jsp'.

Pages : **1** | 2 | 3 | 4

Latest in DLinks

Open Source Web Frameworks' Mailing List Traffic
Trim Spaces in your JSP's HTML Redux
Introduction to scripting in Java, Part 2
Generating JavaDoc For a Project In NetBeans IDE
How to set General options in NetBeans IDE
Toolbar Settings And Shortcuts In NetBeans IDE
Techniques for Complex HQL
Hibernate Search 3.0 Beta 4: new features bandwagon
With Direct Web Remoting (DWR) unnecessary complexity is a bug
Why Would a Groovy Swing Programmer Need a Matisse-like GUI Builder?

Introduction to Java Server Faces

by Shunmuga Raja
19/05/2007

4) Different Phases in a JSF Application

It is very important to understand how a JSF application is initiated by a typical client such as a Web Browser along with the set of phases involved. Most of the different phases in a JSF Web Application is taken care by the JSF framework and only a minimal amount of burden is imposed on the application developers. The six different phases that are involved in a JSF application are as follows,

- View Restoration
- Applying the Request Values
- Validation of User Inputs
- Updating the Model objects
- Execution of Application Logic
- Rendering the Response to the Clients

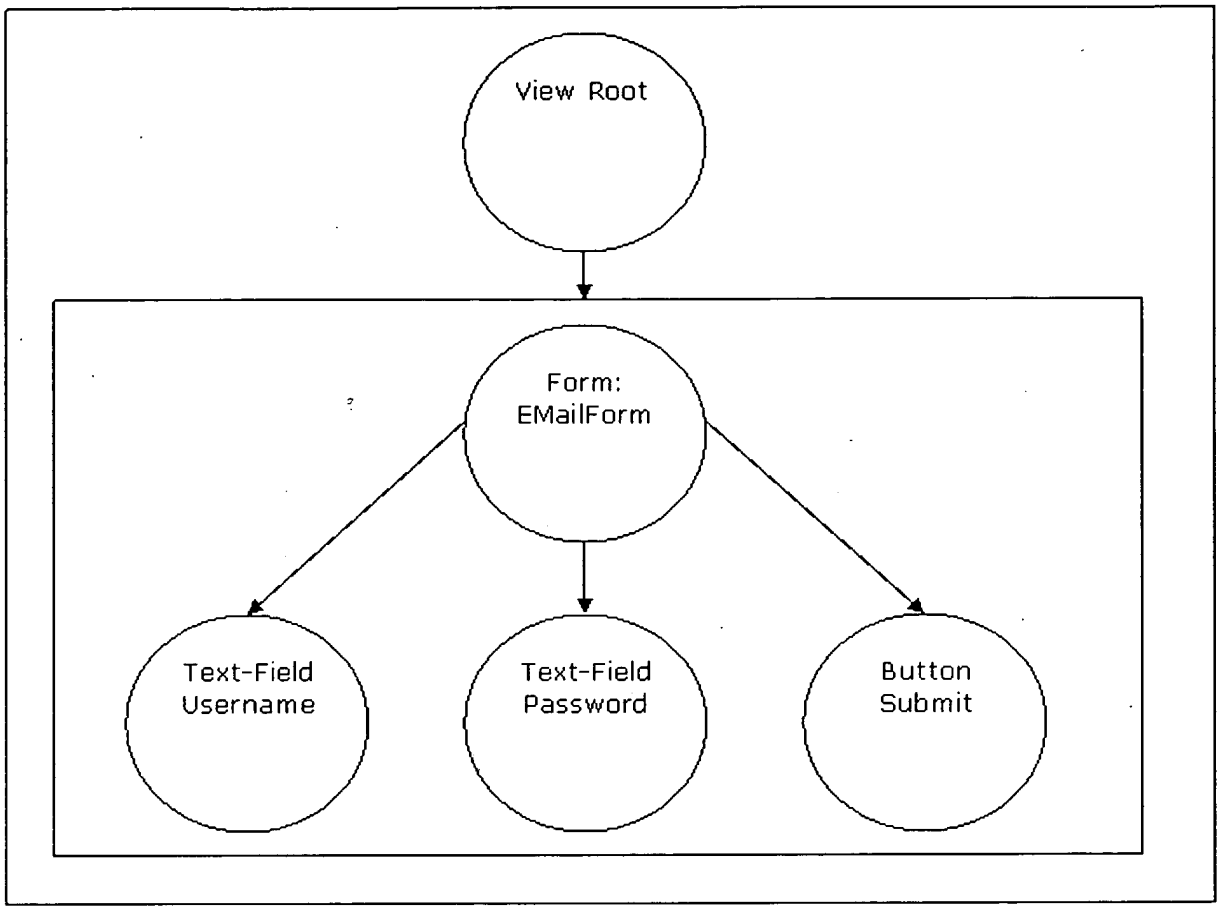
Let us have a brief discussion about the several phases one by one.

4.1) View Restoration

Before getting into details of the various set of actions that happens in this phase, let us define what a ***View*** is in JSF terms. In JSF terminology, a View represents the ***tree of UI components***. Assume a typical e-mail application, which is asking the user to enter the details like email-id and the password. Also there is a submit button which is there to carry the request information from the client to server after the user has filled-in the necessary details.

Technically, there is a form in the page with 3 controls, 2 for the Text-Field holding the email-id and the password and 1 for the Action button. In this context, a ***View*** represents the tree of components that are available in the page. The view along with the components is stored in the server so that the state of the UI components is automatically managed.

In our case, the View may look like this,



Representation of View which points to a Tree of UI Components

Before understanding what the view restoration phase will do, let us see the various components and interactions involved in a JSF Web-Application. In a Web Application, thousands and thousands of requests may come from the clients. Each and every different request may display different page to the user. Every page often has an entirely different ***View (Tree of UI Components)*** associated with it. And each view within the page may have separated ***UI Components*** which may have ***Validators***, ***Converters***, ***Listeners*** and ***Managed Beans***.

Every ***JSF Request*** is often associated with a context called ***JSF Context***. The JSF Context stores the entire ***graph of view***. The primary responsibility of this ***View Restoration*** phase is to identify the ***Current View*** and

to restore this current view from the JSF Context. If no such view is found in the JSF Context, then this phase will create a ***new View*** which hold all the related UI components and associates all the different elements and then places it in the JSF Context.

Upon client request, the View may either be newly created or it can be restored from the current JSF Context, so this phase has been termed as ***View Restoration*** phase.

4.2) Applying the Request Values

View Restoration Phase has ended and the client has been presented with a nice view wherein which he/she is ready to input the appropriate values. After the client has entered the input data, the data is set or it is mapped to corresponding ***UI Component***.

Taking the sample example that we had in the ***View Restoration*** Phase,

The First component is the Text-Field component with some unique identifier, say "emailIdTextField" and the next Text-Field component is represented by the identifier "passwordTextField". Whenever the user submits the values by clicking the submit button, JSF will iterate over the tree of components, pick up every component and sets the corresponding values to the right component that are submitted by the User.

So, after the end of this phase, all the values that are submitted by the client will be set to the appropriate ***UI Components*** taken from the View with the help of the ***Faces Context***.

4.3) Validation of User Inputs

In this phase, all the inputs that are submitted by the client user will be validated for their correctness. JSF will iterate over all the UI components in the ***UI Tree*** and will call the ***Default Validators*** as well as any of the ***Custom Validators*** that have been provided to the component.

Assume that in the email-id text field, user is allowed to enter only in the following format, `username@somedomain.com`. For this type of application specific validation, we can define something called `EMailValidator` to check whether the user input matches the correct email format.

Either kind of validation is possible, ***Direct validation*** which is dedicated only for the particular component as well as ***Delegated Validation***. The result of this phase may end up in the next phase ***Updating the Model Objects*** being called or it may direct JSF to the last phase which is the ***Rendering Response Phase***. Assume that the user has entered an invalid email-id in which a case, the application will display appropriate ***Error Messages*** back to the user. It means that some kind of Error-Response has been given back to the user which is

nothing but the ***Render Response Phase***.

It is worthwhile to mention that two possible set of actions can occur in this phase apart from validations. One is ***Pre-Conversion Operation*** and the other one is the ***Post-Event Operation***.

As mentioned, all the input values from the client are sent to the server only as string values. Some kind of objects called ***Converters*** must be there to convert whatever input the client sent. Say a date value in the client UI has to be converted into a Date object before proceeding with the processing. Even there is a possibility that ***Events*** may be triggered as a result of values being changed; in such a case ***Value-Changed Events*** will be fired.

4.4) Updating the Model objects

Managed Beans form the ***Model Objects*** in a JSF application. Assume that in the mail application, we have represented a form containing two text-fields namely email-id and password. For this, we may have written a Bean class called `UserInfoBean` encapsulating the values of email-id and password.

The next question that arises in mind immediately is who will take care of mapping the two Text-Field values to the two String properties inside the UserInfo bean. The answer has become very simple because of ***EL expressions***. An UI component can be mapped directly against a ***Java Bean Property*** declaratively in the definition of the UI component.

For example, consider the code following code snippet,

```
<html:inputText id = "emailIdTextField" value = "#{UserInfoBean.emailId}">
</html:inputText>
```

The above code declares an input component of type 'Text-Field' with an identifier 'emailIdTextField'. It has one more important attribute called ***value***. This attribute is populated with a string called `#{"UserInfoBean.emailId"}`. This is essentially an ***EL Expression*** which tells to associate the value of the email-id text-field to the property name called `emailId` in the class `UserInfoBean`. Synchronizing an ***UI Component Value*** with that of a ***Bean Property*** is as simple like this.

4.5) Execution of Application Logic

If you remember that in the second phase ***Apply Request Values***, we saw that as soon as the values from the

input fields are applied to the corresponding UI components events will be generated. Only the events will be generated in that phase. The generated event won't be notified to all the listeners who have registered for the events. The events in the second phase will be notified to all the listeners (if any) in this phase only. Remember that in JSF, all the UI components often have their ***Default Listeners*** associated with them.

Take the example of a Button Control. This UI component, by default will have an `ActionListener` associated with it. Whenever user clicks this button control, an event called action event will be generated. For example, if the model Bean Object has been registered to receive the events, then the corresponding event handler will be called in this phase

Consider the following snippet code,

```
<html:commandButton
    id = "submitButton"
    value = "Submit the Form"
    actionListener = "#{UserInfoBean.submitButtonClicked}">
...
</html:commandButton>
```

In the above example, a button control with the display names 'Submit the Form' has an attribute called ***actionListener*** with value `#{UserInfoBean.submitButtonClicked}`. As mentioned before, this is a ***JSF EL Expression***, which will resolve to a method called `submitButtonClicked` inside a bean class `UserInfoBean` as follows

```
class UserInfoBean{
    public void submitButtonClicked(javax.faces.event.ActionEvent event){
            // Process code here.
    }
}
```

As we can see from the above snippet code, the `submitButtonClicked()` method is passed an `ActionEvent` (note, the event is of type `javax.faces.event.ActionEvent` and not the AWT `ActionEvent` found in `java.awt.event.ActionEvent`). One useful method in the `ActionEvent` class is the `getComponent()` method which will return the component that generated this event.

4.6) Rendering the Response

Now, it's time to render (display) the response back to the client application in the ***Rendering the Response*** phase and this is the last phase in the life-cycle. One more thing to note is that before sending the response data to the client, JSF implementations store the View (which represents the Tree of UI components) in the ***Faces Context object***, so that it can be restored back by the ***View Restore*** back when the request comes to the same page.

In JSF, there is a clear separation between the data representation and the display of the UI Components. More specifically, the UI components that we talked about don't have the implementation of how they can be displayed. They will just represent the properties and the behaviors of the control. The rendering is handled separately by ***Renderer Components*** in JSF in the form of ***Renderer Kits***.

Assume that there are two types of client application, one is the traditional HTML Browser which can understand and interprets HTML, and the other is the WML Browser which can interpret only WML. So, they will be two separate rendering kits namely the ***HTML Rendering Kit*** and the ***WML Rendering Kit***. Sun's JSF implementation comes with the ***Standard HTML Rendering Kit***. The entire API for HTML rendering is available in the `javax.faces.component.html`. Third party vendors can seamlessly plug-in the rendering Kits for WML (or for any other kind of display).

Events in Life-cycle phases

It is not that only ***UI Components*** can fire ***Events***. Even events are emitted by the ***Request Processing Life-cycle Phases***. To be precise 6 different life-cycle phases comes into the picture as soon as a client request comes to a JSF application. During every life-cycle phase, events are emitted by the JSF during the beginning and the ending of that particular phase. Following code snippet will demonstrate this,

```
class MyViewRestorePhaseListener extends PhaseListener{

    public void beforePhase(PhaseEvent event){
            //Do application specific initialization code here.
    }

    public void afterPhase(PhaseEvent event){
            //Release the object resources here.
    }

    public PhaseId getPhaseId(){
            return PhaseId. RESTORE_VIEW;
    }
}

LifecycleFactory factory = (LifecycleFactory)
        FactoryFinder.getFactory(FactoryFinder.LIFECYCLE_FACTORY);
Lifecycle lifecycle = factory.getLifecycle(DEFAULT_LIFECYCLE);
lifecycle.addPhaseListener(new MyViewRestorePhaseListener());
```

The above code essentially obtains the current `Lifecycle` object and then adds a new view restore phase listener to the Lifecycle. So whenever a ***View Restore*** phase is about to happen, JSF will call the `beforePhase()` method passing the `PhaseEvent` object. Applications can make any initialization related logic within the beforePhase() method. And when the phase has ended, the `afterPhase()` method is called by the JSF Implementation.

Pages : 1 | **2** | 3 | 4

Latest in DLinks

Open Source Web Frameworks' Mailing List Traffic
Trim Spaces in your JSP's HTML Redux

Introduction to scripting in Java, Part 2
Generating JavaDoc For a Project In NetBeans IDE
How to set General options in NetBeans IDE
Toolbar Settings And Shortcuts In NetBeans IDE
Techniques for Complex HQL
Hibernate Search 3.0 Beta 4: new features bandwagon
With Direct Web Remoting (DWR) unnecessary complexity is a bug
Why Would a Groovy Swing Programmer Need a Matisse-like GUI Builder?

Pages : 1 | 2 | **3** | 4

Introduction to Java Server Faces

by Shunmuga Raja
19/05/2007

5) JSF Tag Libraries

Tags in JSP terms often represent a kind of task or an action that can do a small piece of work. ***Repeated Actions*** can be encapsulated within a tag and can be used in various places. A set of tags represent a ***Tag library***. In JSF, two categories of tag libraries are available. They are

- JSF Core Tag Libraries
- JSF HTML Tag Libraries

Let us see the two different Tag Libraries one by one.

5.1) JSF Core Tag Libraries

These are set of libraries that provide tags which are responsible for categorizing the various UI elements in the form, to provide validation and conversion stuffs, tor manage set of event listeners for a particular UI component and so on. The declaration of the JSF Core Tag library is given below,

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="face" %>
```

In the above snippet, the prefix can be anything like 'face' or 'f' or anything that doesn't matter. Some of the commonly used tags inside the core tag library are given as follows, (they have been categorized based on their functionality)

- View
- actionListener, valueChangeListener
- validateRange, validateLongRange, validateDoubleRange
- convertDateTime, convertNumber

The ***view*** tag is the top level view and it acts as a container for all the UI components in a form. The entire tree of UI components are logically contained in this ***view*** tag. ***actionListener*** and ***valueChangeListener*** are kind of Event-Listener tags. They can be associated with UI components. Whenever an action is made on a UI component an event is triggered by a UI component and the event will be notified to all ***Registered Listeners***. Tags that are related to validations are ***validateRange***, ***validateLongRange*** and ***validateDoubleRange***. They can be used to restrict the number of characters that are typed in an input component and also can be used to check whether the given number is between certain range. ***convertDateTime*** and ***convertNumber*** represent the conversion tags which are used to convert the user input values to some object representation.

5.2) JSF HTML Tag Libraries

These set of tags are used to display the UI components in a ***HTML-aware Browser***. Do remember that HTML Browser is not the only target client for a JSF application. It can even be a ***WML Browser*** for a mobile phone in which case they will be a need for a separate ***WML Renderer Kit***. The HTML Tag Libraries can be classified as follows,

- input and output tags (text-field and text-area)
- Command tags (Buttons and Hyper-links)
- Selection tags (check-box, radio-button, drop-down box)

The input tags may represent the normal ***Text-Field*** or ***Text-Area*** to get the input data and it can even be a

password text-field. The command tags represent the ***Actionable Tags*** like the click of the ***Button*** or the ***Hyperlink***. Multiple inputs for a UI component can be represented by ***Check-Box*** and ***Radio-Button*** controls.

6) Configuration File

A ***JSF Configuration File*** is a XML File which can be used to configure the various components like the Managed Beans, Validators, Converters, Navigation Rules etc. A sample configuration file is given below,

```
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
"http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

    <managed-bean>
        <managed-bean-name>testBean</managed-bean-name>
        <managed-bean-class>test.TestBean</managed-bean-class>
    </managed-bean>

    <navigation-rule>
    <description>Navigation for the test page.</description>
    <from-view-id>/test/test.jsp</from-view-id>
    <navigation-case>
        <from-outcome>test</from-outcome>
        <to-view-id>/test/response.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>

    <validator>
        <validator-id>TestValidator</validator-id>
        <validator-class>test.TestValidator</validator-class>
    </validator>

    <converter>
        <converter-id>TestConverter</converter-id>
        <converter-class>test.TestConverter</converter-class>
    </converter>

</faces-config>
```

The above configuration file declares the various elements like managed beans, navigation rules, Validator and converters.

The managed bean for a JSF application is defined in the configuration file using the ***managed-bean*** element type, within which the name of the bean, the fully qualified class name of the bean along with some descriptive information can be declared. A configuration file can contain any number of managed beans. There is an element called ***managed-bean-scope*** which defines how long this bean object is valid. Possible values for scope are session, request and application.

Static Navigation between web pages can be declared within the configuration file with the help of ***navigation-rule*** and ***navigation-case*** elements. Any number of ***navigation-case*** can follow the ***navigation-rule*** elements.

User-defined ***Validators*** and ***Converters*** can be made available in a JSF Application through the use of ***validator*** and ***converter*** tags. They are often identified by a unique identifier along with the fully qualified class names with the help of ***id*** and ***class*** tags.

Pages : 1 | 2 | **3** | 4

Latest in DLinks

Open Source Web Frameworks' Mailing List Traffic
Trim Spaces in your JSP's HTML Redux
Introduction to scripting in Java, Part 2
Generating JavaDoc For a Project In NetBeans IDE
How to set General options in NetBeans IDE
Toolbar Settings And Shortcuts In NetBeans IDE
Techniques for Complex HQL
Hibernate Search 3.0 Beta 4: new features bandwagon
With Direct Web Remoting (DWR) unnecessary complexity is a bug
Why Would a Groovy Swing Programmer Need a Matisse-like GUI Builder?

Introduction to Java Server Faces

by Shunmuga Raja
19/05/2007

7) Sample Application

Let us write a sample application called User Registration system. The system will provide a User Interface that will collect various values from the user such as name, username, password, birthday and phone-number.

7.1) JSF Configuration File

The following is the ***faces-config.xml*** file used in the sample application which will contain all the necessary configurations that are related to the declaration of managed beans, Custom Validators for validating the user inputs and a Simple Navigation Rule for Page Navigation.

```
<?xml version='1.0' encoding='UTF-8'?>

<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

    <managed-bean>
        <managed-bean-name>UserBean</managed-bean-name>
        <managed-bean-class>user.registration.UserBean</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>

    <navigation-rule>
        <description></description>
        <from-view-id>/UserRegistration.jsp</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/Success.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>

    <validator>
        <validator-id>PhoneNumberValidator</validator-id>
        <validator-class>
                    user.registration.validator.PhoneNumberValidator
        </validator-class>
    </validator>

</faces-config>
```

7.2) Web.xml file

Following is the ***web.xml*** which has to be located in the WEB-INF directory of the application root. As we can see from the ***servlet-mapping*** element, whatever Request Uri's which follows the pattern ***/faces/**** will be directed to the `FacesServlet` available in the `javax.faces.webapp` package.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>

</web-app>
```

7.3) UserRegistration.jsp file

The UserRegistration.jsp page represents the user interface wherein which the user is allowed to enter the various inputs like name, username/password, and birthday and phone number.

All the ***HTML UI Components*** have to be embedded with the ***<f:view>*** and the ***</f:view>*** tags. Text-Field, password field and the button object are represented by ***<h:inputText>***, ***<h:inputSecret>*** and ***<h:commandButton>*** respectively. Every individual control has been assigned a unique identifier and all the input values for the components are made mandatory by setting the ***required*** attribute to 'true'. The *element is used to display an* ***Error Message*** *if the user didn't provide any value for the field. The* ***for*** *attribute in the* ***message*** *element identifies for which element the error message has to be displayed.*

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>

<html>

    <head>
        <title>User Registration</title>
    </head>

    <body>

    <h1>User Registration</h1>

    <f:view>
        <h:form>

            <p>Enter your name:
            <h:inputText value="#{UserBean.name}"
                         id="nameTextField" required="true"/>
            <h:message for="nameTextField"/>
            </p>

            <p>Choose a username:
            <h:inputText
                    value="#{UserBean.userName}"
                    id="userNameTextField"
                    required="true"/>
            <h:message for="userNameTextField" />
            </p>

            <p>Enter the password:
            <h:inputSecret
                    value="#{UserBean.password}"
                    id="passwordTextField"
                    required="true"/>
            <h:message for="passwordTextField" />
            </p>

            <p>Enter your birthday:
            <h:inputText
                    value="#{UserBean.birthday}"
                    id="birthdayTextField"
                    required="true">
```

The various input components are tied directly to the properties of the managed Java Bean called the `UserBean`*. Since the birthday is a Date field, the string entered by the user is converted to a Date object with the help of the converter tag* ***<f: convertDateTime>****. Also a simple validation is provided for the phone number field with the help of the* ***<f:validator>*** *tag. The logical output of this page is 'success' which is represented by the* ***action*** *attribute of the 'commandButton' element. This value has to be used to define the* ***from-outcome*** *element that is present within the* ***navigation-case*** *element.*

7.4) Success.jsp

This success.jsp is the response page for the 'UserRegistration.jsp' page. The page is just meant to display the values that are entered by the user along with a confirmation message telling that the user has been successfully registered. Displaying of message is done with the help of ***<h:outputText>*** *tag.*

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>

<html>
    <head>
        <title>Registration Successful</title>
    </head>
    <body>

    <h3>Registration Successful</h3>

    <P>
    <f:view>
    <h:form>
        <P>The following information has been saved succesfully<P/>
        <P>Name:         <h:outputText value = "#{UserBean.name}" /> <P/>
        <P>User Name:    <h:outputText value = "#{UserBean.userName}"/> <P/>
        <P>Birthday:     <h:outputText value = "#{UserBean.birthday}"/> <P/>
        <P>Phone Number: <h:outputText value = "#{UserBean.phoneNumber}"/> <P/>
    </h:form>
    </f:view>
    </P>

    </body>
</html>
```

7.5) UserBean.java

Following is the code for the `UserBean` *class which is the managed Java Bean as being defined in by the* ***managed-bean*** *element in the* ***faces-config.xml****. This class is used to just hold the various input values with the properties and the behaviours according to the Java Bean Specification.*

```
package user.registration;

import java.util.Date;

public class UserBean {

    private String name;
    private String userName;
    private String password;
    private Date birthday;
    private String phoneNumber;

    /** Creates a new instance of UserBean */
    public UserBean() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public Date getBirthday() {
        return birthday;
    }
```

7.6) PhoneNumberValidator.java

Validation for the input component phone-number Text-Field is taken care by this class. This class provides a ***Simple Validation Rule*** *to check the validity of the phone number. It first checks whether the input entered by the user is a number by parsing the string object. If the parsing fails because the user has entered some character or invalid data, then a* `ValidationException` *is thrown. Then it just checks whether the number of digits is 10. If not, then also a* `ValidationException` *is thrown.*

```
package user.registration.validator;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class PhoneNumberValidator implements Validator{

    public PhoneNumberValidator() {
    }

    public void validate(
            FacesContext context, UIComponent component, Object value)
                throws ValidatorException {
        String strValue = (String)value;
        checkForNumbersOnly(strValue);
        if (strValue.length() != 10){
            throwException("Number of phone digits must be 10");
        }
    }

    private void checkForNumbersOnly(String strValue){
        try{
            long phoneNumber = Long.parseLong(strValue);
        }catch (Exception exception){
            throwException("All Phone Digits must be of numbers only.");
        }
    }

    private void throwException(String errMessage){
            FacesMessage message = new FacesMessage();
            message.setDetail(errMessage);
            message.setSummary(errMessage);
            message.setSeverity(FacesMessage.SEVERITY_ERROR);
            throw new ValidatorException(message);
    }
}
```

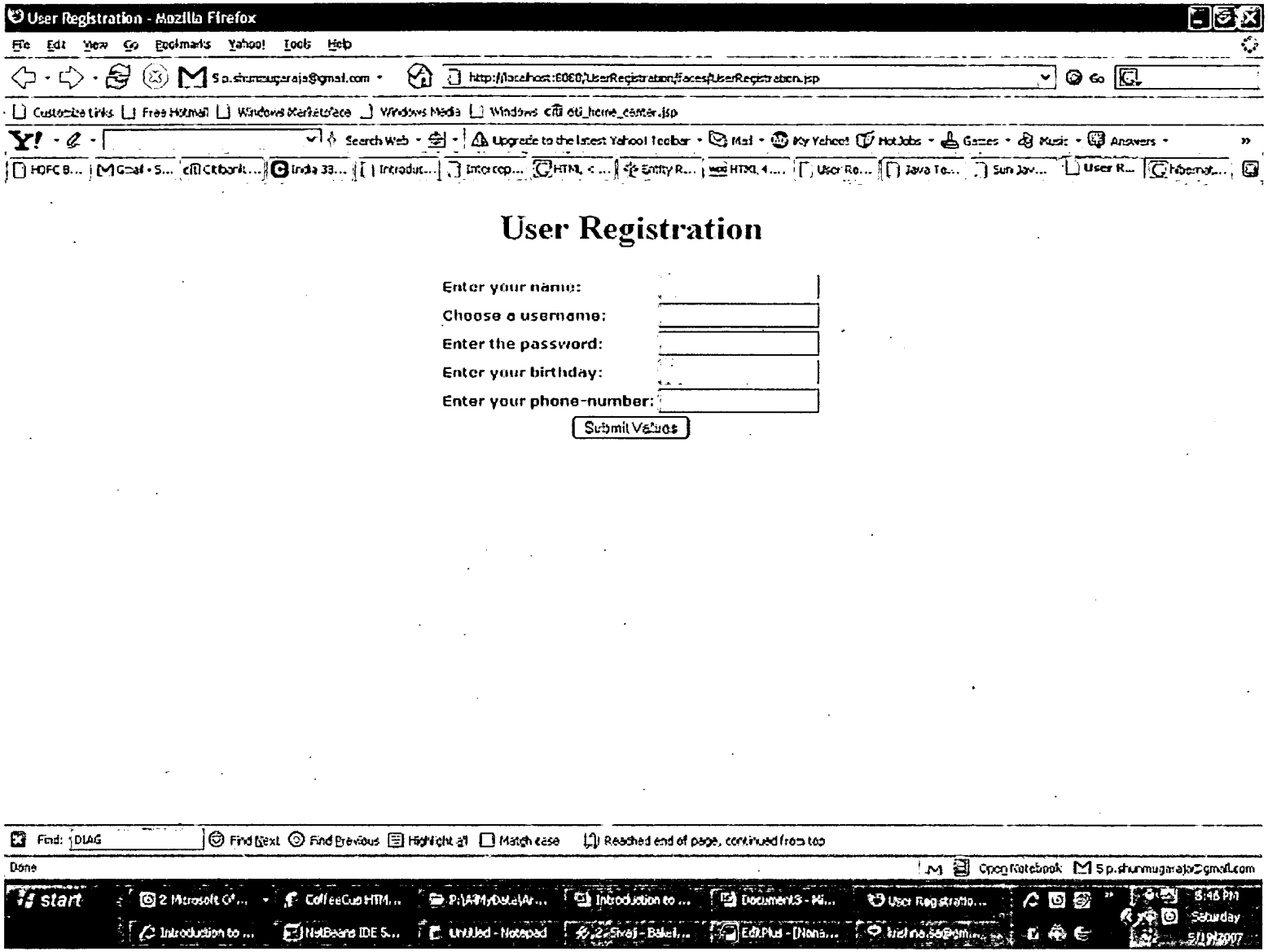
7.7) Running the Application

Following are the pre-requisite softwares needed to run the application.

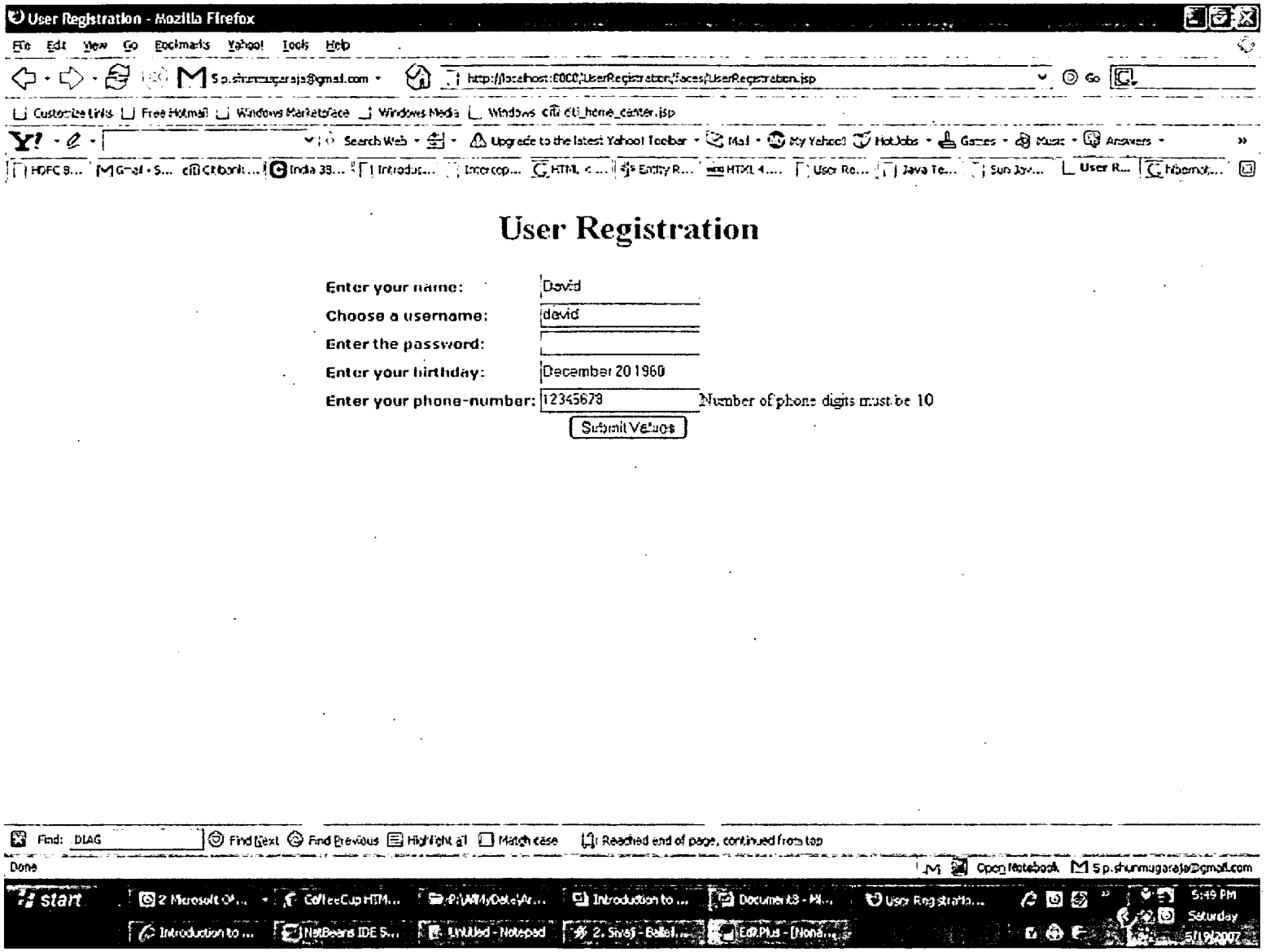
- *A web container like Tomcat (http://www.ip97.com/apache.org/tomcat/tomcat-6/v6.0.13/bin/apache-tomcat-6.0.13.zip)*
- *Java Development Kit (http://java.sun.com/javase/downloads/index_jdk5.jsp)*

Create a directory called UserRegistration and copy all the JSP files (UserRegistration.jsp and success.jsp) into it. Next, create a directory within UserRegistration with name WEB-INF and then copy the web.xml and JSF Configuration file (faces-config.xml). Create a sub-directory called 'classes' under WEB-INF directory and copy all the class files ('UserBean.class' and 'PhoneNumberValidator.class') along with the directory structure. Make sure that 'jsf-impl.jar' is placed inside the lib directory which is inside the WEB-INF directory.

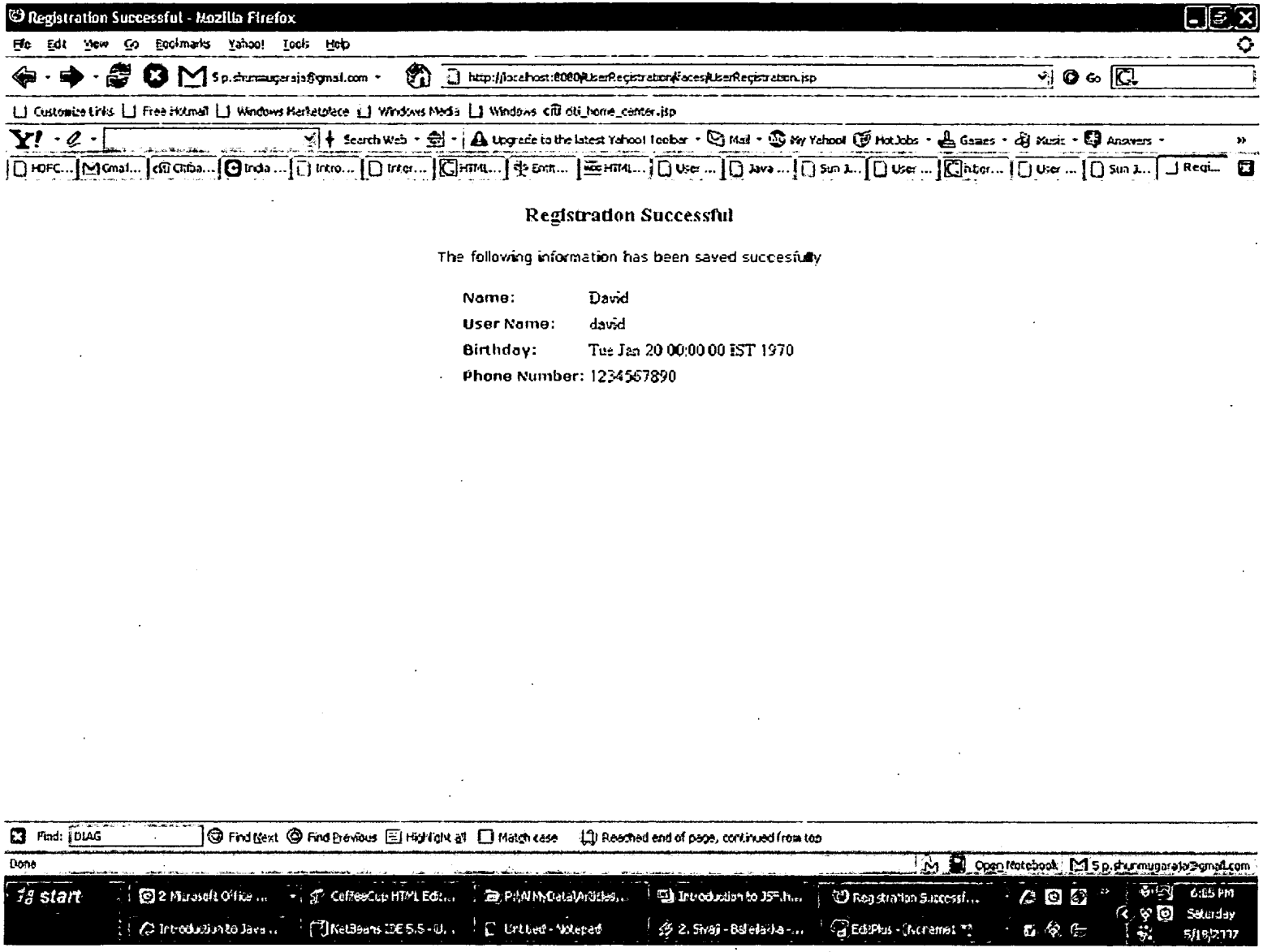
Copy the UserRegistration directory to the Tomcat's root directory and then re-start the server. Point the URL to 'http://localhost:8080/UserRegistration/UserRegistration.jsp' to see the application running.



Initial screen-shot that presents the user Interface for getting the various inputs from the User.



Phone Number validation has taken place, as the number of digits entered by the user is only 9.



Final output page that displays a confirmation message telling that the user profile has been saved successfully.

Pages : 1 | 2 | 3 | 4

Latest in DLinks

Open Source Web Frameworks' Mailing List Traffic
Trim Spaces in your JSP's HTML Redux
Introduction to scripting in Java, Part 2
Generating JavaDoc For a Project In NetBeans IDE
How to set General options in NetBeans IDE
Toolbar Settings And Shortcuts In NetBeans IDE
Techniques for Complex HQL
Hibernate Search 3.0 Beta 4: new features bandwagon
With Direct Web Remoting (DWR) unnecessary complexity is a bug
Why Would a Groovy Swing Programmer Need a Matisse-like GUI Builder?

App # 10/628959

MyEclipse Visual JSF Designer Quickstart

Table of Contents

1. Introduction

In this tutorial we are going to cover some of the features available in the new Visual JSF Designer available in MyEclipse 5.5 and later.

While MyEclipse has offered a Visual JSP Designer in the past, the advanced Visual JSF designer now available in MyEclipse 5.5 and later offers many new features to JSF developer like dynamic analysis of build path to determine the taglibs available on the palette, and rendering of advanced components, e.g. dataTables and panelGrid. For a more general overview of all the MyEclipse Visual JSF Designer features see the MyEclipse Visual JSF Overview document.

2. Suggested Audience

This tutorial is intended for developers who are somewhat familiar with MyEclipse or Eclipse so you recognize navigation within the IDE, and understand some of the more common views like the debugger. This tutorial also assumes knowledge of JSF page development in general.

To learn more information about the topics presented in this tutorial please have a look at the links in our Resources section. To get a better feel for MyEclipse and learning more about it, please check out our product Documentation for more material.

3. System Requirements

This tutorial was created with MyEclipse 5.5. If you are using a another version of MyEclipse, most of these screens and instructions should still be very similar.

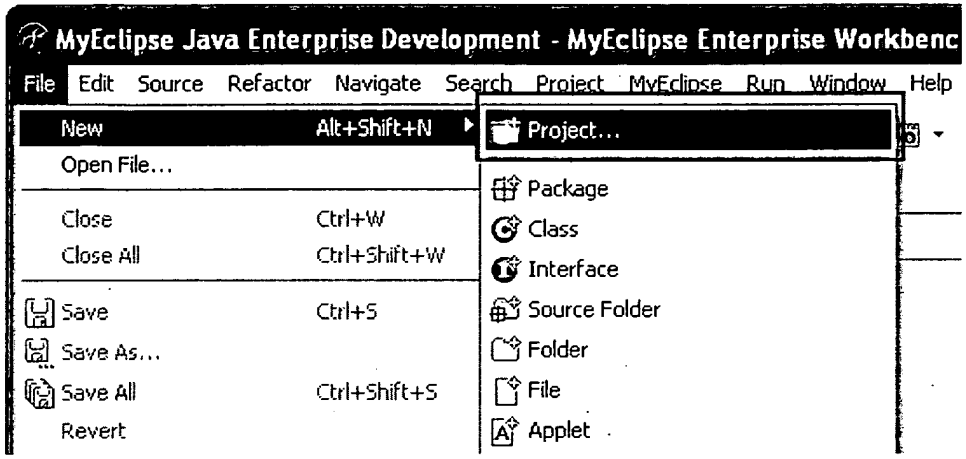
If you are using a newer version of MyEclipse and notice portions of this tutorial looking different than the screens you are seeing, please let us know and we will make sure to resolve any inconsistencies.

4. Getting Started

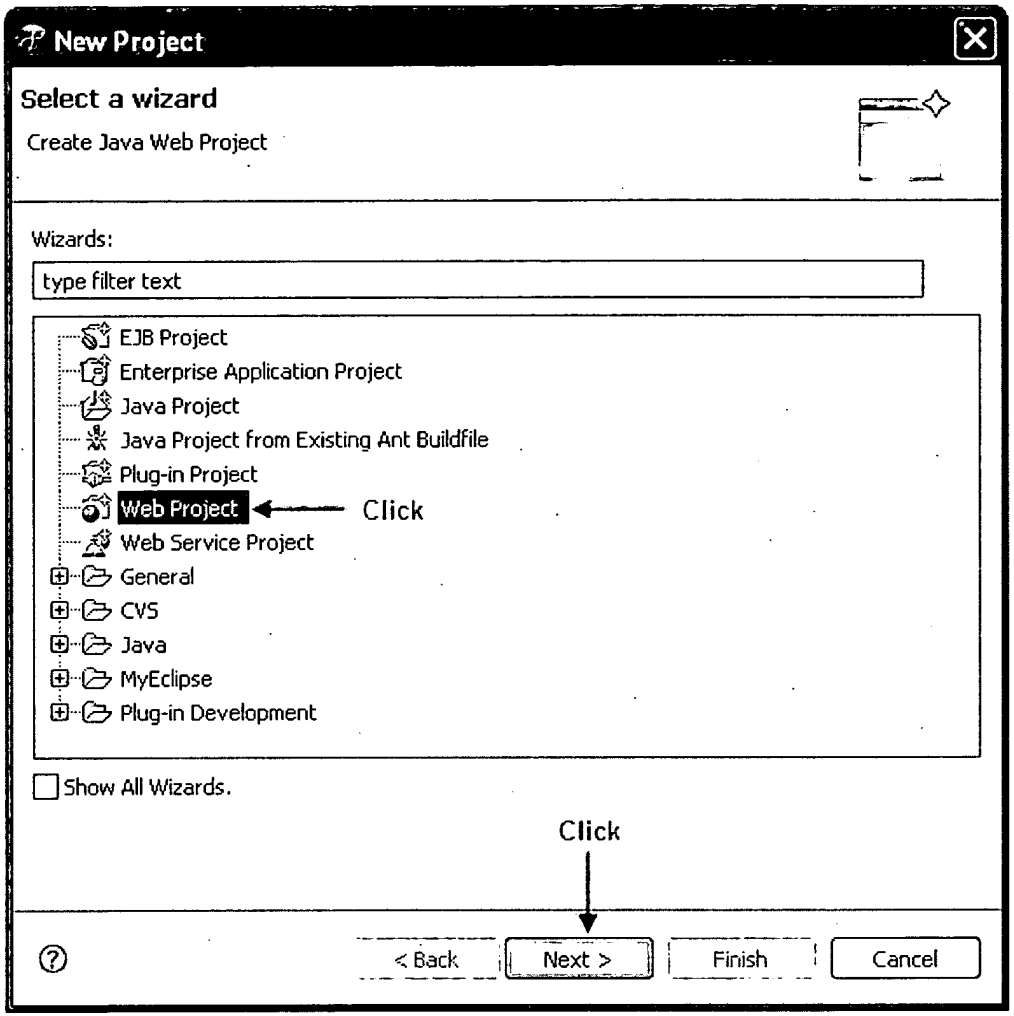
To get started with the new Visual JSF Designer you must first have a *Web Project* that has *JSF Capabilities* added to it. From there you can create new JSF pages or edit existing ones using the visual designer. The designer recognizes JSF pages by the tag libraries imported and used on the page, so it's possible to use the designer with the different JSF page formats.

Creating a Web Project

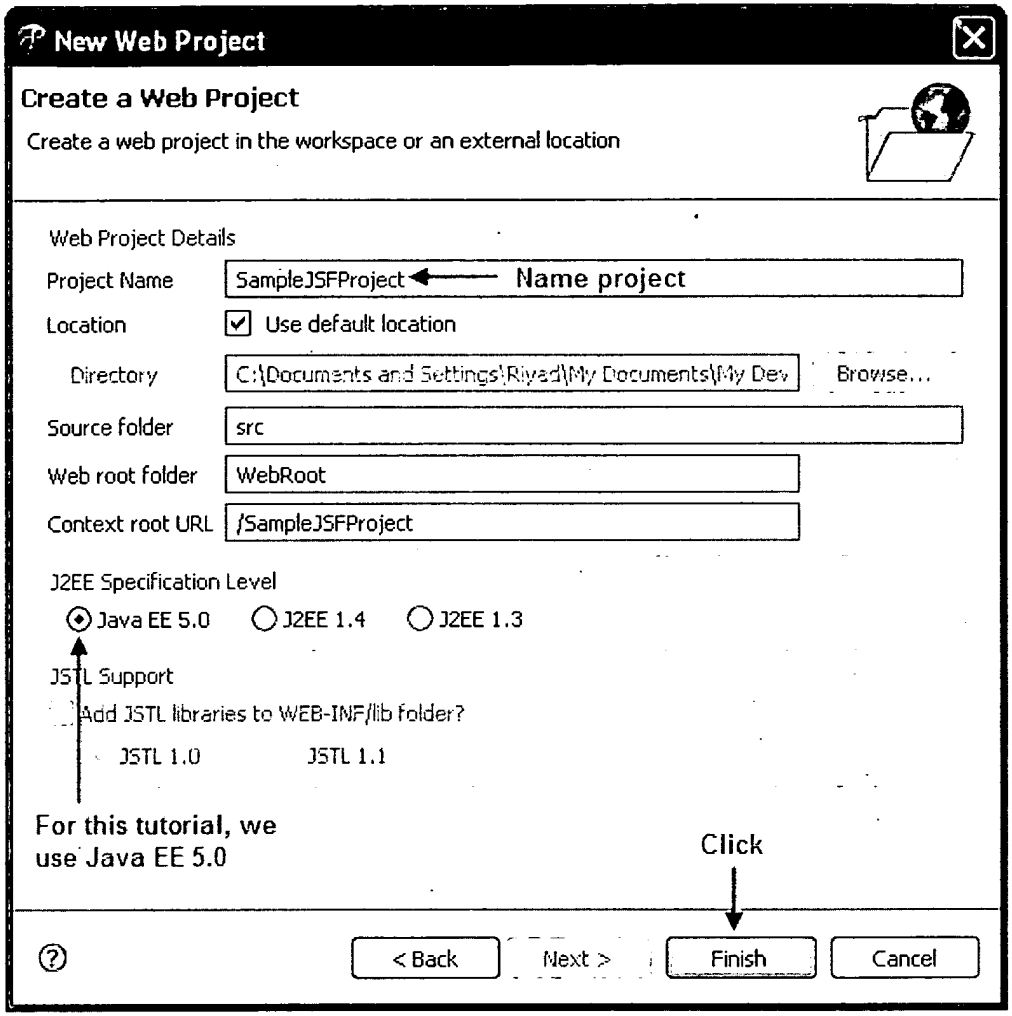
To begin, we need to first create our *Web Project*:



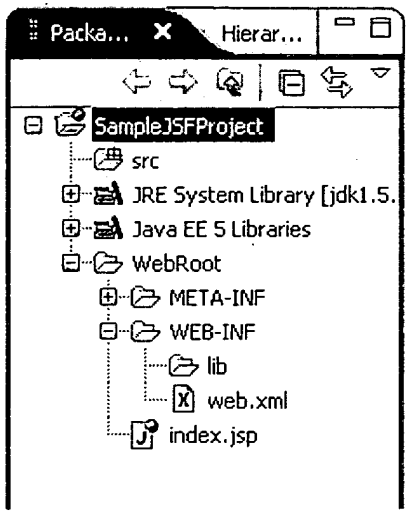
Then select *Web Project* and hit *Next*:



Give the project a name, and in this tutorial we select the Java EE 5.0 specification to use for our project (which includes the JSTL and JSF JARs as part of the specification):



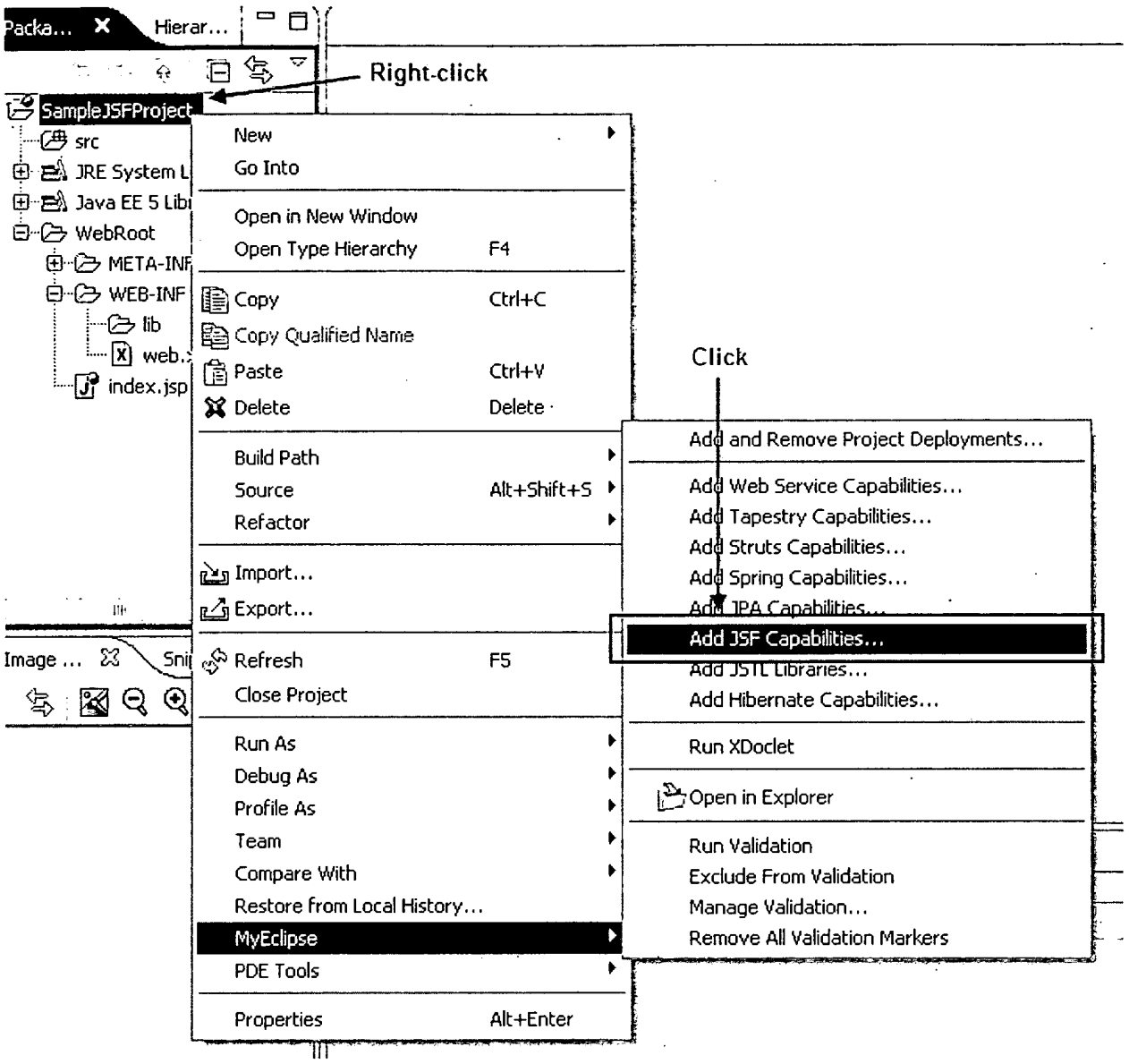
After MyEclipse has created the new project, it will look something like this:



Even though Java EE 5.0 contains the JSF libraries, we still need to add *JSF Capabilities* to our project so MyEclipse knows to treat the project like a JSF application.

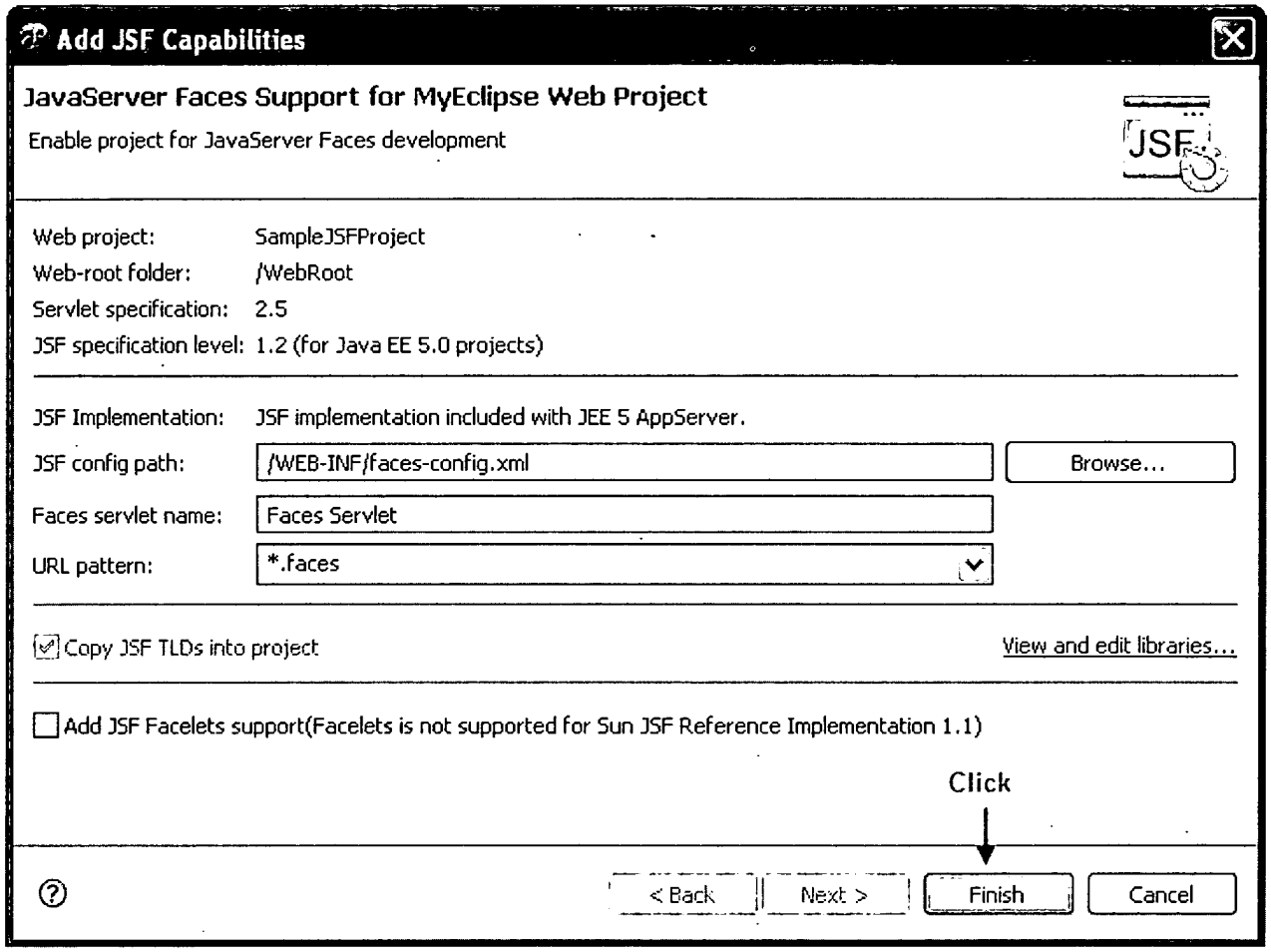
Adding JSF Capabilities

We add *JSF Capabilities* to our project just like we add other capabilities. Start by right-clicking on the project, and going down to *MyEclipse* then selecting *Add JSF Capabilities*:



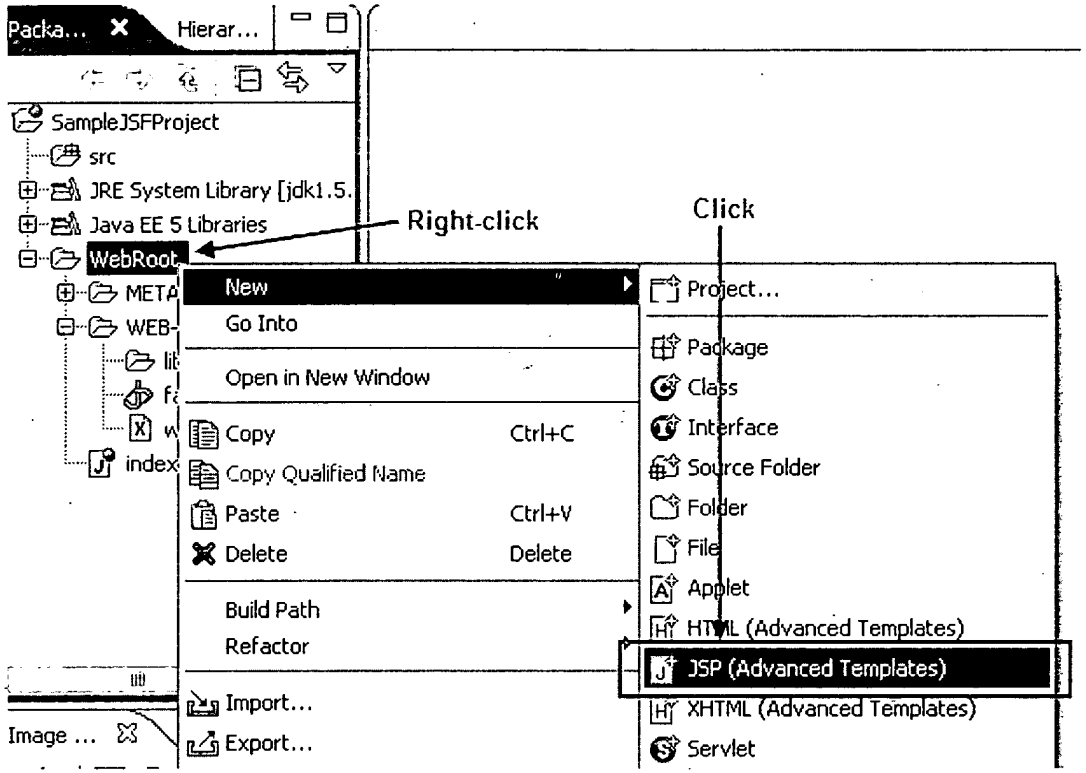
When the wizard pops up, with a Java EE 5.0 project, there are no library selections to make (e.g. Sun RI, MyFaces, etc.) because the JSF

implementation is included with the libraries in your application server. Besides choosing a few of the default mappings and possibly adding **Facelets** support, you can just click *Finish*:

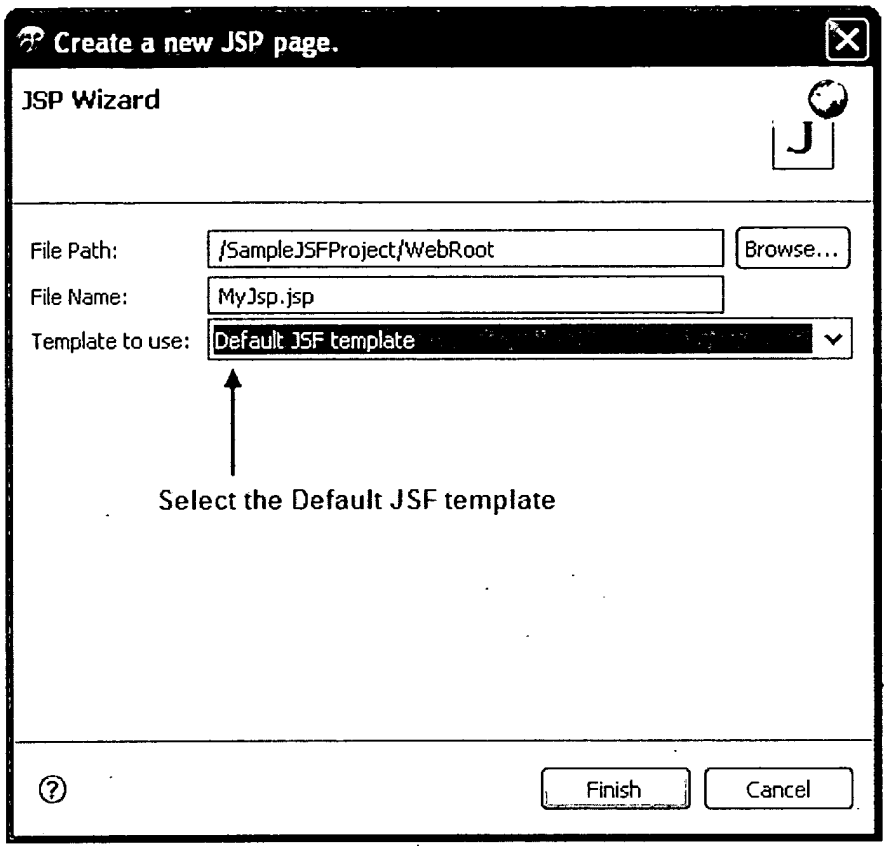


5. Creating a JSF Page

Once the project is setup, you can create your first JSF Page and start building it with the Visual JSF Designer. Right-click on the folder you wish to add your page to, and select *New* then *JSP (Advanced Templates)*:

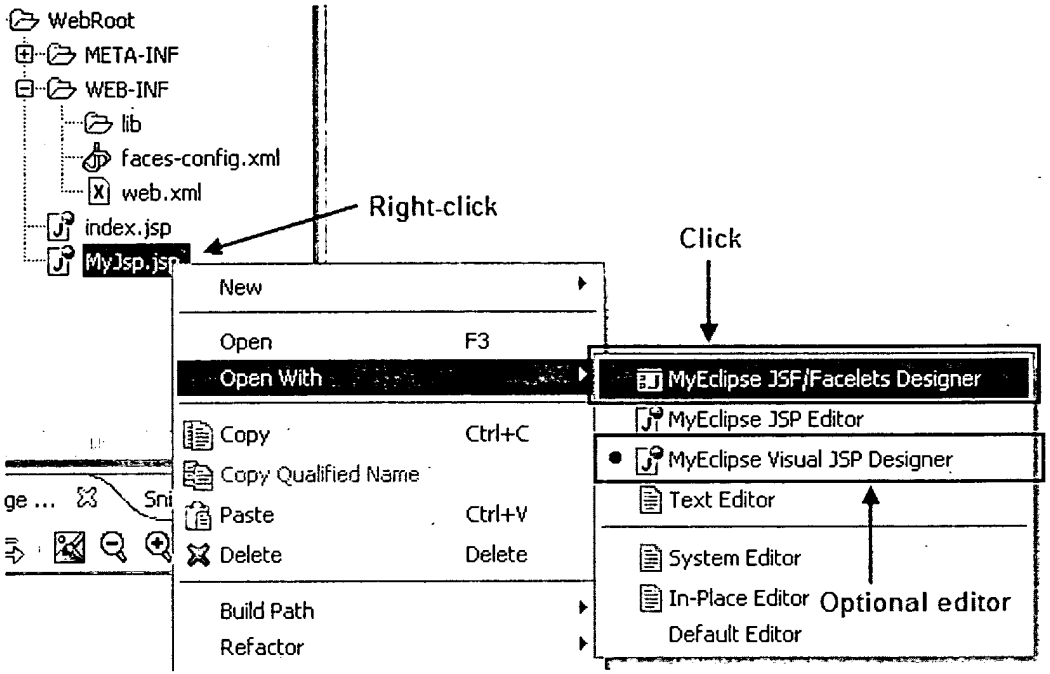


Then be sure to select the *Default JSF template* for the page template to use:

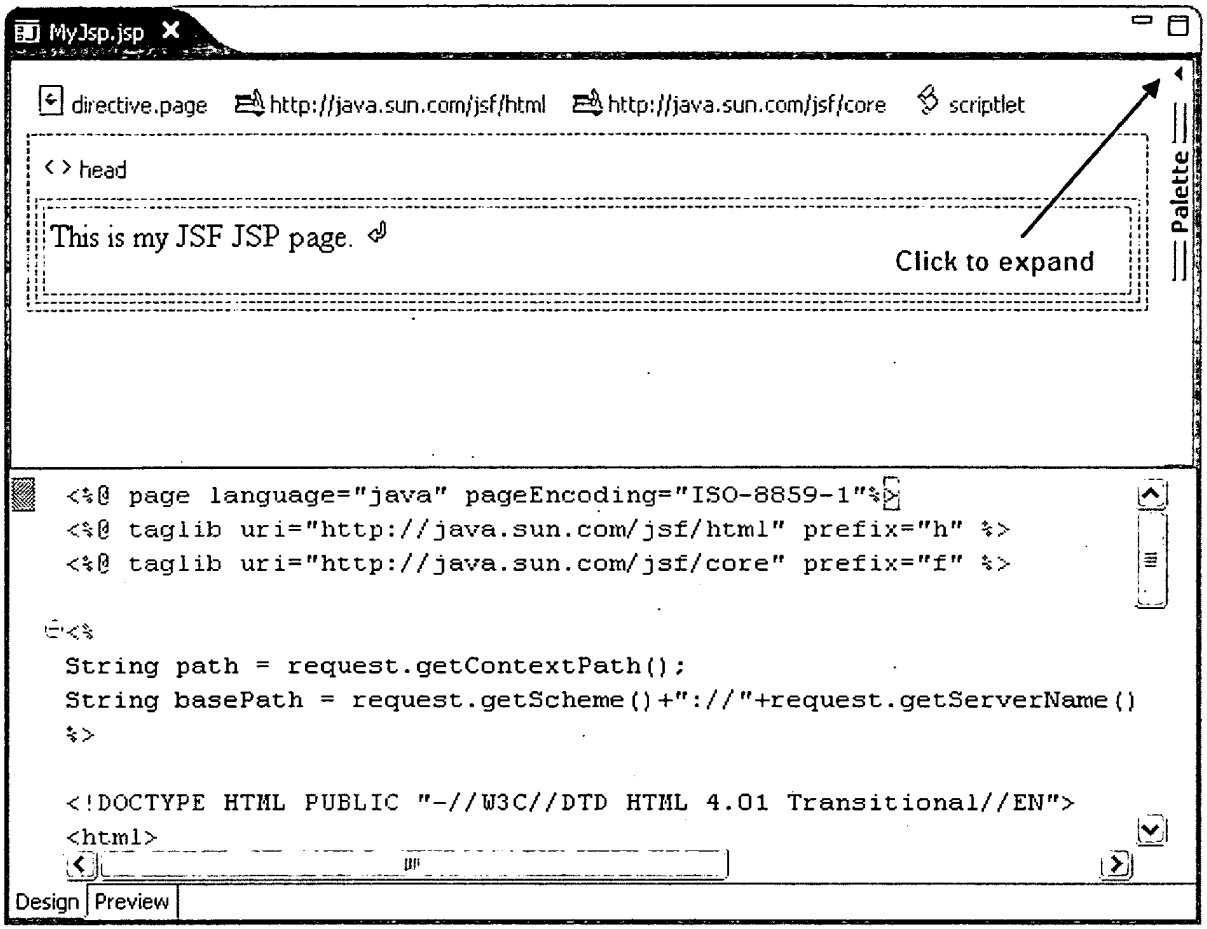


When the new page opens up in the designer, you can close it. We want to right-click on the new JSF page, and see our available editors to get a better idea of what we can edit this page with.

If you select the file that you just created and right-click on it, you will see the following:



Notice that this file can be edited with any of these editors. The two visual editors are the *MyEclipse Visual JSP Designer* and the newer *MyEclipse Visual JSF Designer*. In this tutorial we are going to open the page with and use the *MyEclipse Visual JSF Designer*, so go ahead and click that. Your page will open up in a designer that looks like this:

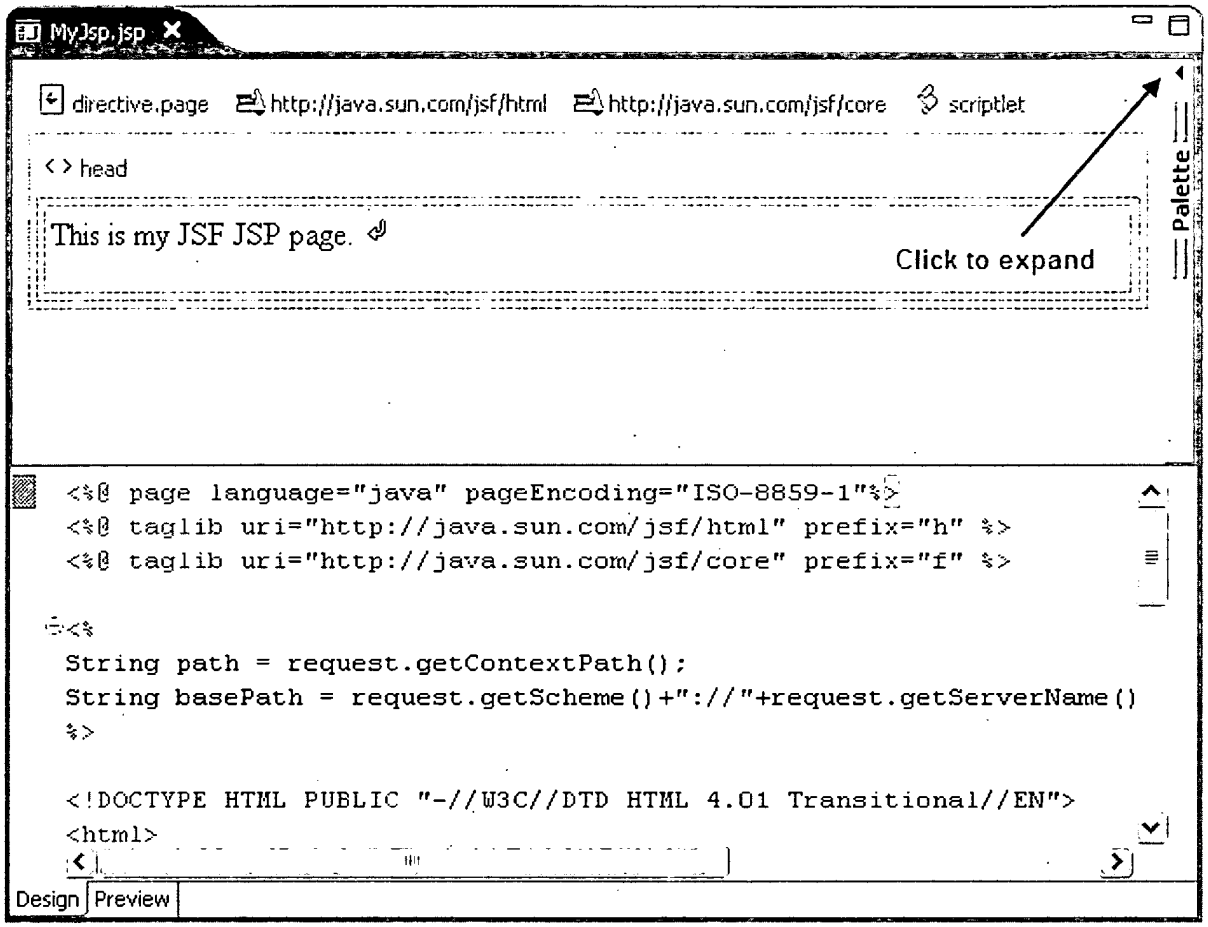


Now that our page is ready and open in the designer, let's start editing it.

6. Designing a JSF Page

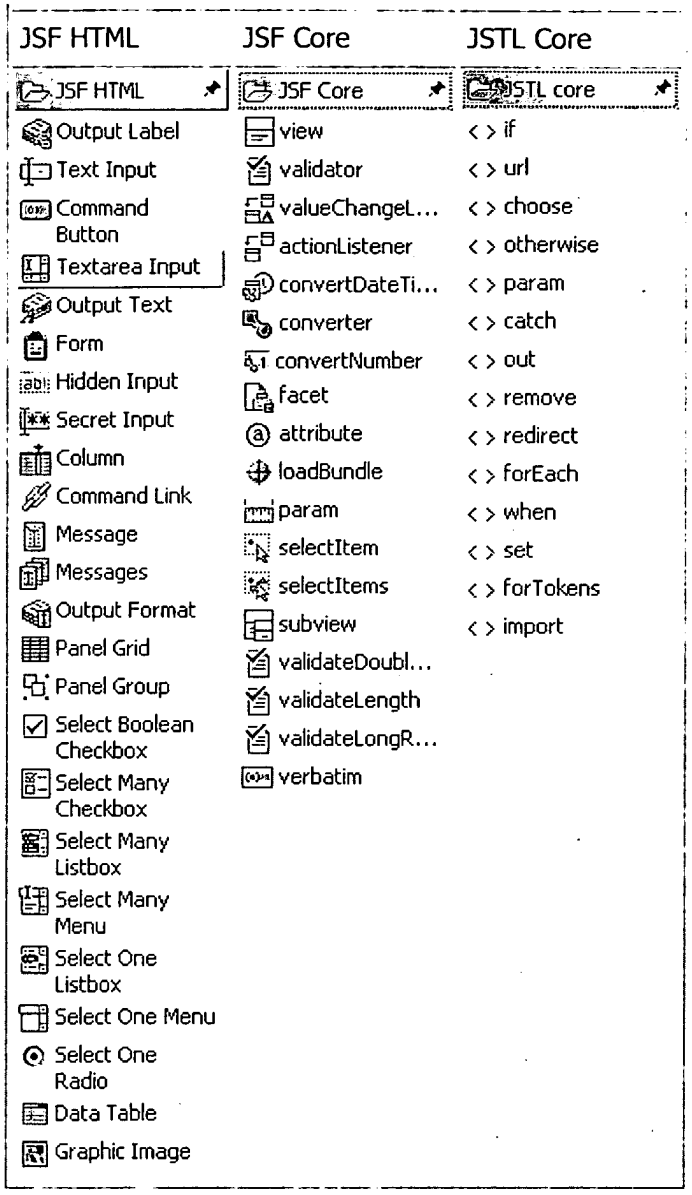
The first thing you want to do after getting the page open in the designer is to expand the palette on the right hand side of the design by

clicking the left arrow on the top of it:



The palette will read the tag libraries out of your build path and load them so you can utilize the Drag and Drop interactions with building your web pages.

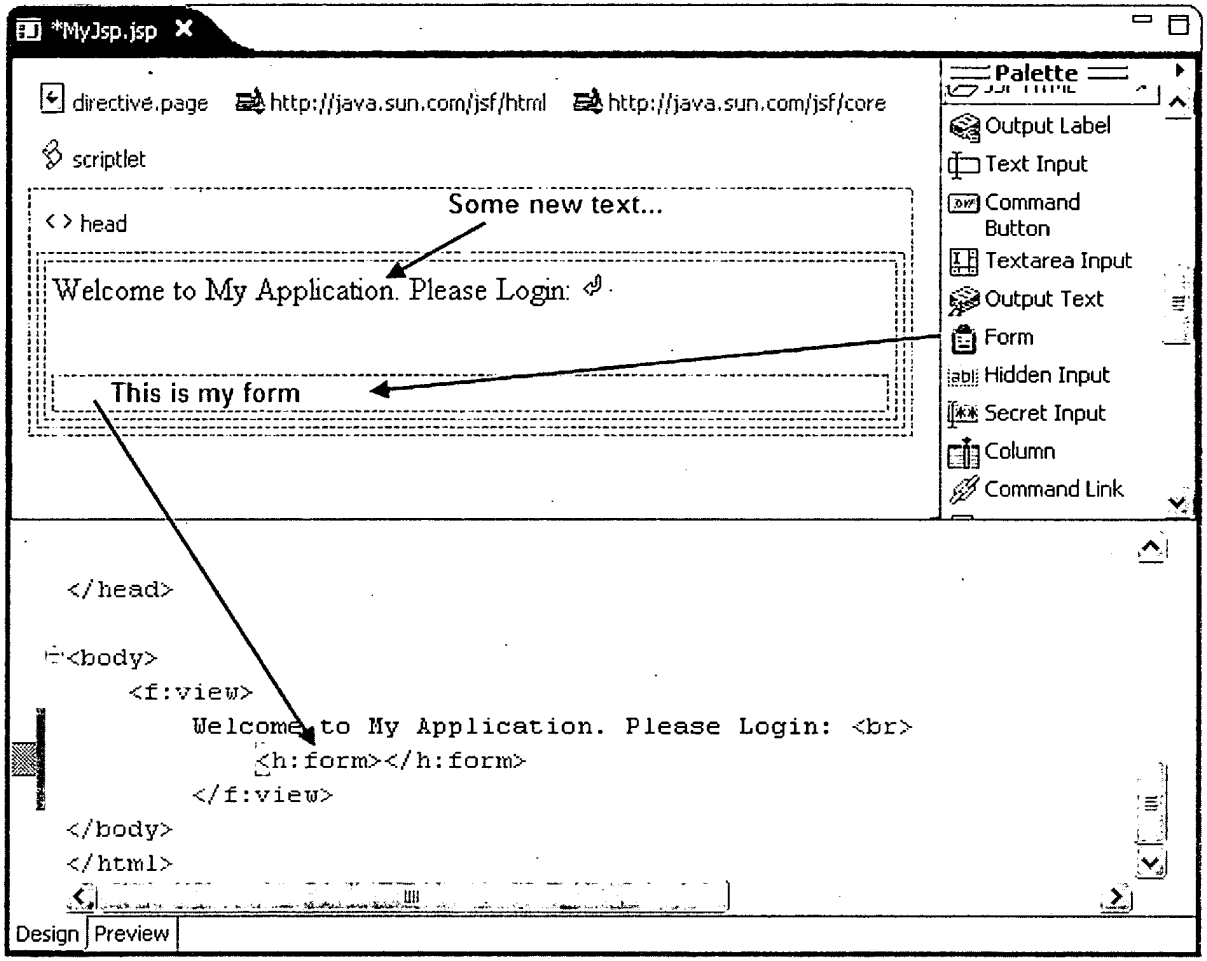
Some of the more important tag groups to be aware of are as follows:



There are the most common sets of tags you will likely be using while designing your pages. Let's use these items to Drag and Drop a login

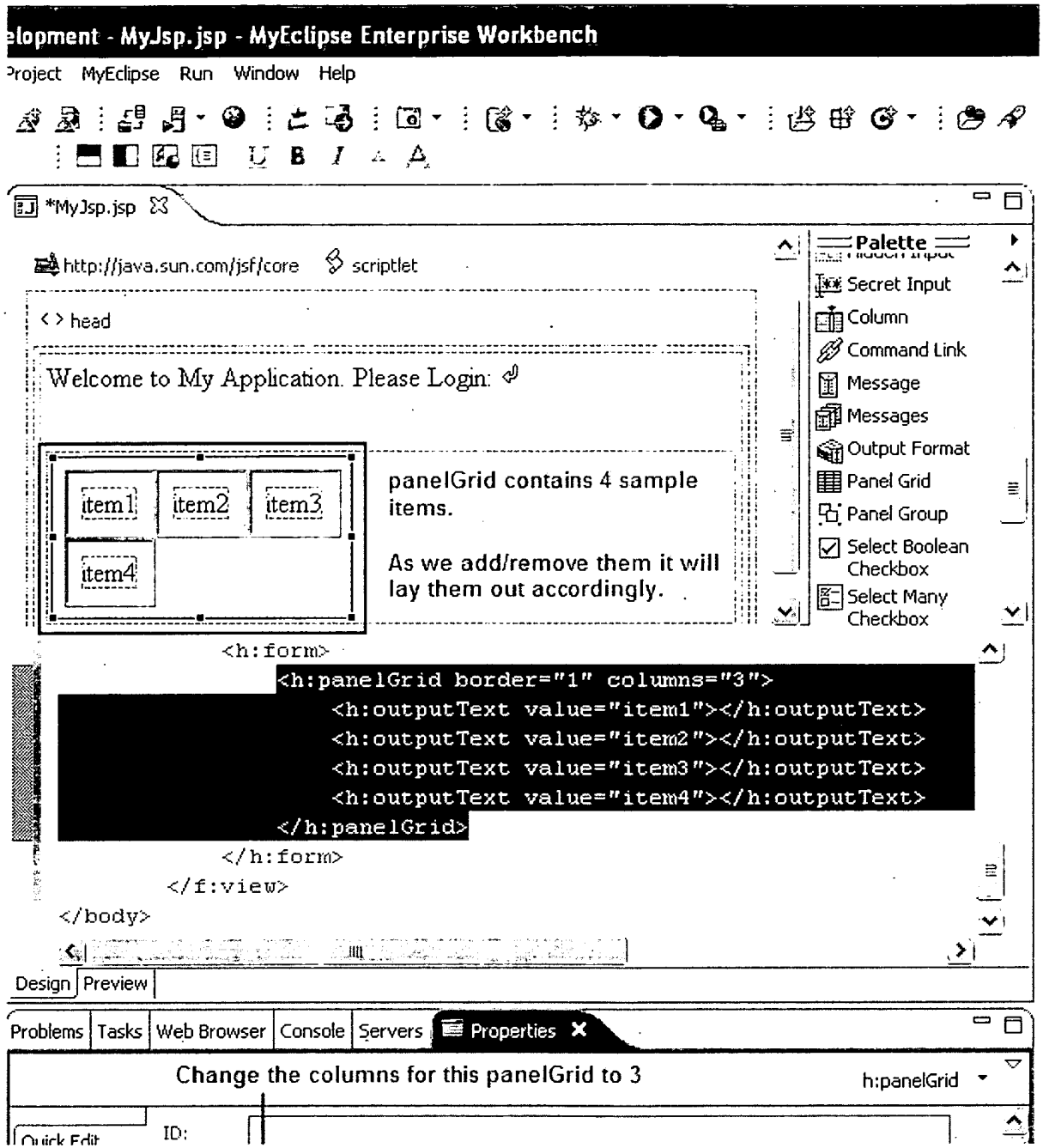
form onto our new page.

First we drag a JSF HTML *form* onto the page and update the page text a bit:



Now we want to layout a typical login form with a username, password and login buttons. To lay these items out nicely, it sounds like it would be a 3x2 table, but if we want to include space for login error messages, we would likely want a 3x3 table.

In JSF there is a component that will layout it's contents in a table automatically for us, it's called a *panelGrid*. Let's go ahead and drag a *panelGrid* into our *form* and be sure to set the *columns* value to 3:

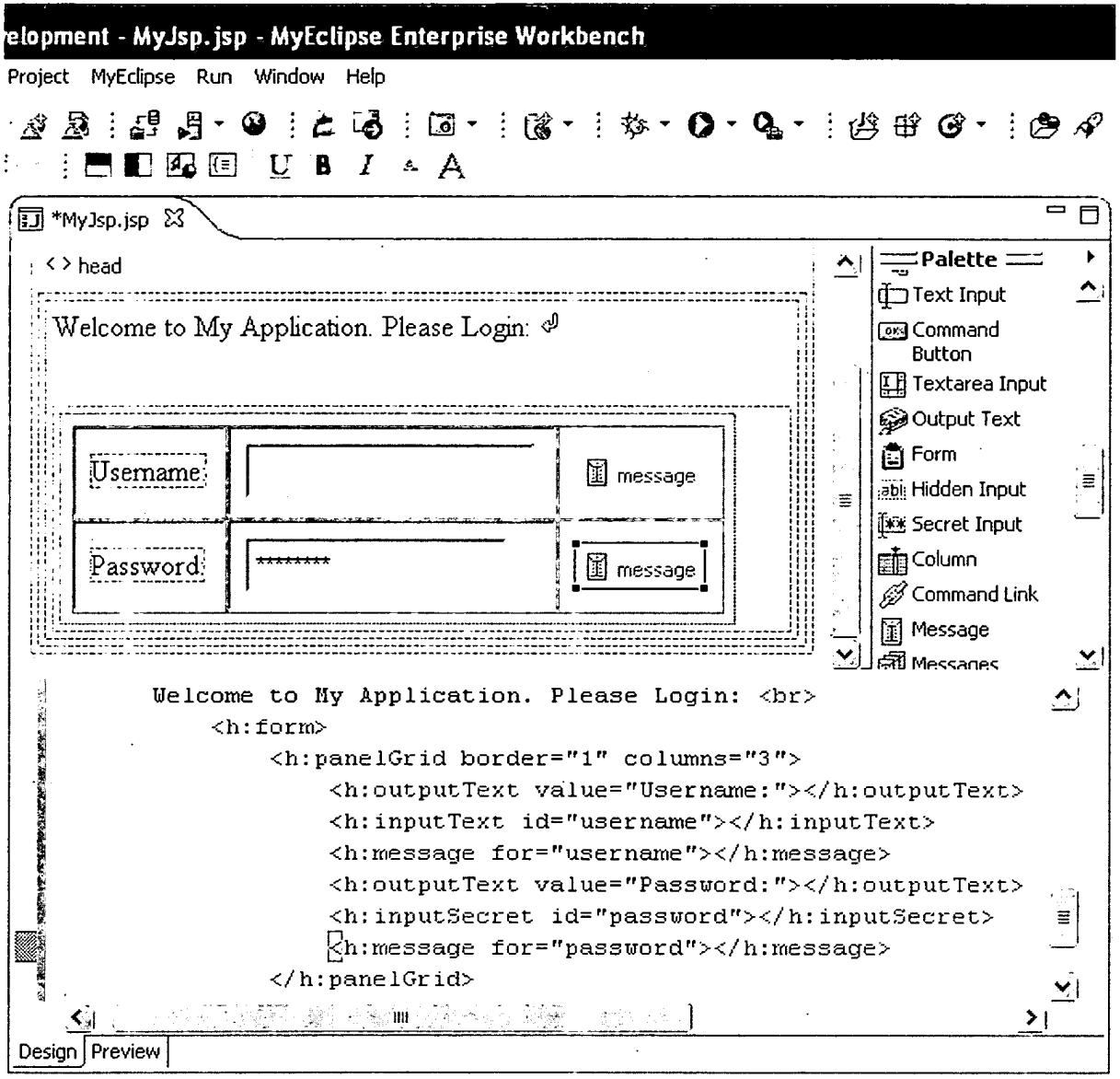

elopment - MyJsp.jsp - MyEclipse Enterprise Workbench
Project MyEclipse Run Window Help
*MyJsp.jsp
http://java.sun.com/jsf/core
scriptlet
<> head
Welcome to My Application. Please Login:
item1
item2
item3
item4
panelGrid contains 4 sample items.
As we add/remove them it will lay them out accordingly.
Palette
Secret Input
Column
Command Link
Message
Messages
Output Format
Panel Grid
Panel Group
Select Boolean Checkbox
Select Many Checkbox
<h:form>
<h:panelGrid border="1" columns="3">
<h:outputText value="item1"></h:outputText>
<h:outputText value="item2"></h:outputText>
<h:outputText value="item3"></h:outputText>
<h:outputText value="item4"></h:outputText>
</h:panelGrid>
</h:form>
</f:view>
</body>
Design
Preview
Problems
Tasks
Web Browser
Console
Servers
Properties
Change the columns for this panelGrid to 3
h:panelGrid
Quick Edit
ID:

The first thing to notice is that when the *panelGrid* is added, the designer automatically adds 4 sample components to it, to get an idea of how output will work.

For this tutorial, we are going to place the following components in the *panelGrid* in the given order:

- outputText: "Username:" label
- inputText: username text field (ID=username)
- message: display username error messages (FOR=username)
- outputText: "Password:" label
- inputSecret: password text field (ID=password)
- message: display password error messages (FOR=password)

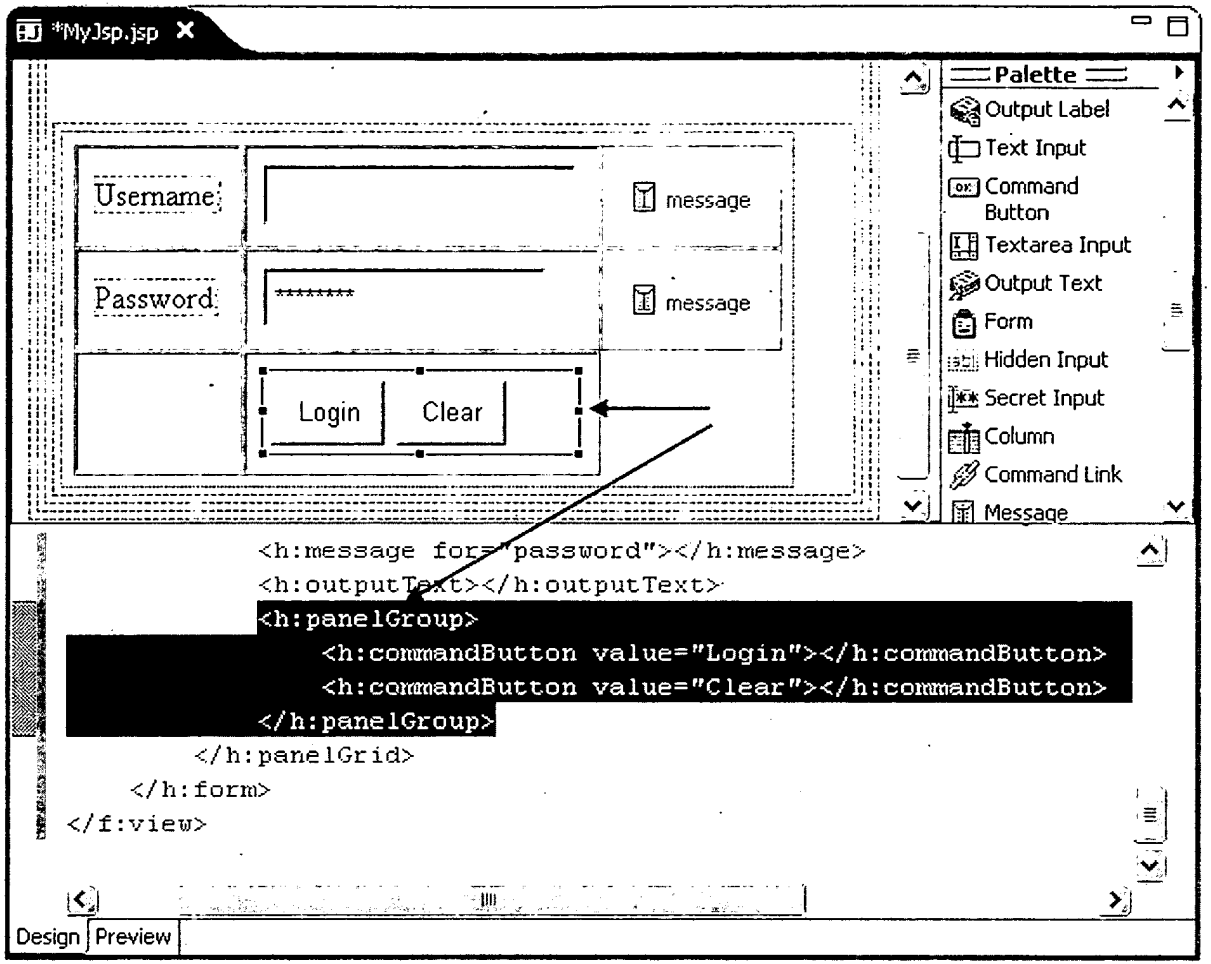
So we haven't added the buttons yet, but at this point our form is done and properly laid out and looks like this:



It's important to note that the *message* components will only be rendered when they have messages to display.

Now let's add our buttons. Given the layout of our login form, we may want to left-align the buttons under the input boxes to make the form look nice. If we simply place a single button in each cell (1 under Password, 1 under *inputSecret*) they are going to be unevenly spaced. However, the way the *panelGrid* works is to take components added directly to it, and lay them out in a table, cell-by-cell. To be able to group our two buttons together, and place them under the input fields, we will need to use a *panelGroup*.

The first thing we need to do is add an empty component (we used an empty *outputText*) to the *panelGrid*, so it places it under the Password label. Second, we need to add a *panelGroup*, so it places it under the input fields. Then inside the *panelGroup*, we will add our two buttons. The result looks like this:



Once that *panelGroup* is added, we don't have to add another component to be placed under the existing *message* components. The *panelGrid* will keep everything laid out correctly.

Now the design portion of our page is done and has given you a good idea of how the designer works. Of course, if you were building a real JSF application, you would need to step back into the page, and using the designer, assign action handlers to the buttons, and value bindings to the input fields to make sure your managed bean was correctly backing the values on this page.

These steps are outside the realm of this tutorial, which was simply intended to introduce you to the new Visual JSF Designer. If you'd like to see how to design a real JSF application, please check out the JSF Tutorial in the MyEclipse help documents or available from the MyEclipse site.

7. Conclusion

We hope you have found this tutorial on the Visual JSF Designer helpful and given you a good starting point to begin using it. The goals of the Visual JSF Designer is to make the component-specific job of creating a JSF application easier to visualize and work with. JSF applications can get complex, and being able to design the pages in a JSF application visually can greatly simplify that task.

If you have any suggestions for us to help make it more informative, please let us know.

Below we would like to provide you with some more information pertaining to the topic covered in this tutorial. We offer the FAQ section for quick references to common questions and the Resources section with links to other helpful resources online that you may want to become familiar. We realize we can't cover every question you may have in one tutorial, but between this tutorial contents and our additional learning resources we hope you are on your way to feeling comfortable with the technology.

8. Resources

In this section we want to provide you with additional links to resources that supplement the topics covered in this tutorial. While this is not an exhaustive list, we do make an effort to point to the more popular links that should provide you with diverse, high-quality information.

- MyEclipse Visual JSF Designer Overview
- MyEclipse LoginDemo JSF Application Tutorial
- Sun JSF Homepage (Lots of Articles and Resources)
- Glassfish JSF Reference Implementation Project Page
- Wikipedia Page for JSF

9. Feedback

We would like to hear from you! If you liked this tutorial, has some suggestions or even some corrections for us please let us know. We track

all user feedback about our learning material in our Documentation Forum. Please be sure to let us know which piece of MyEclipse material you are commenting on so we can quickly pinpoint any issues that arise.